CSC4005 2023 Fall Project 3 Report

Tianci Hou 121090184

1 Distributed/Parallel Sorting Algorithms with MPI

1.1 Set up

Environment: Refer to course GitHub page.

Type	Compilation flags
Sequential	`-O2`
OpenMP	`-O2 -fopenmp -mavx2`
MPI	`-O2 -fopenmp -mavx2`

(a) Compilation Flags

```
$ cd path/to/project
$ mkdir build && cd build
$ cmake ..
$ make -j4
$ sbatch ../src/sbach.sh
$ sbatch ../src/sbach_bonus.sh
```

(b) Build Commands

Figure 1: Compilation and Build Information

1.2 Overall performance

Workers	QuickSort(M)	BucketSort(M)	Odd-Even-Sort(M)	MergeSort(T)	QuickSort(T)
1	12961	10205	38794	12780	9951
2	12116	7886	28698	12760	9939
4	8564	4928	18279	6663	5360
8	7272	3437	10669	3753	2997
16	7315	2601	5674	2361	1921
32	8069	2475	2804	1726	1598

Table 1: Overall performance. (M: MPI, T: Threads)

1.3 Speed up and efficiency

Workers	QuickSort(M)	BucketSort(M)	Odd-Even-Sort(M)	MergeSort(T)	QuickSort(T)
1	5.76%	14.35%	-4.43%	99.61%	30.70%
2	2.76%	12.87%	1.29%	60.12%	9.46%
4	6.18%	3.63%	1.77%	58.17%	45.60%
8	9.72%	0.58%	8.36%	47.54%	52.35%
16	9.56%	-0.69%	30.74%	46.00%	98.23%
32	9.67%	-0.24%	111.09%	19.52%	125.09%

Table 2: Speed up[1] compared with baseline

[1] *Since the cluster pressure during testing is relatively high, it may be slightly slower than the baseline.*

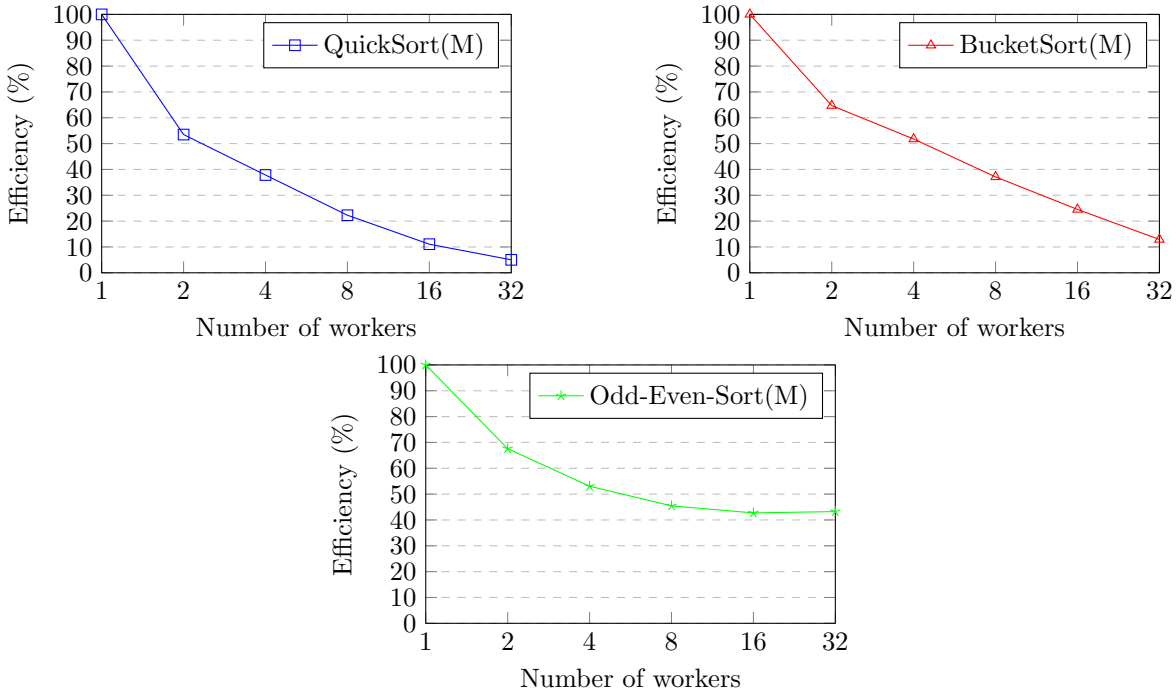

Figure 2: Efficiency in normal part

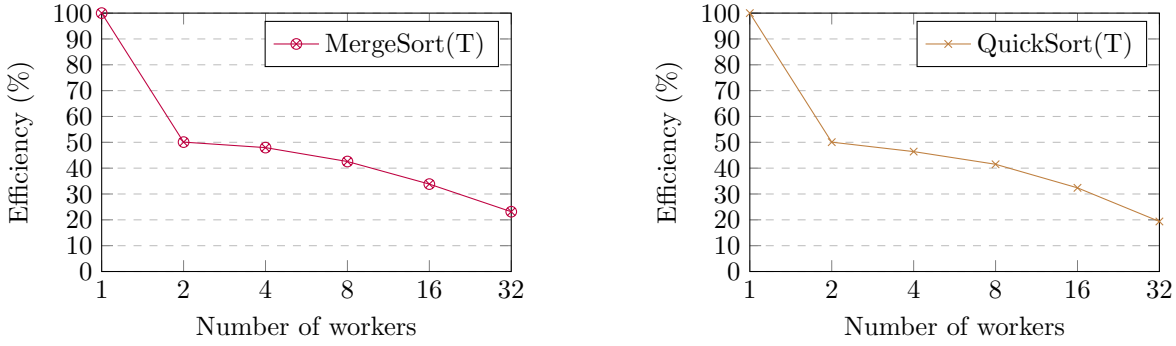

Figure 3: Efficiency in bonus part

1.4 Correctness

I write a simple shell script to test all of my implementations. Also I generate 50 random test cases (different length). The outputs including bonus part are same as the `std::sort`.

2 Task 1: Process-Level Parallel Quick Sort with MPI

2.1 Solution

According to the hints on the page of GitHub, we first split the input vector into many parts, and each worker does a QuickSort for that part. Then we do the merge operation.

2.2 K-way merge problem

We need special data structure for solving this problem. Since this course is about parallel programming, I will not write too much about data structure.

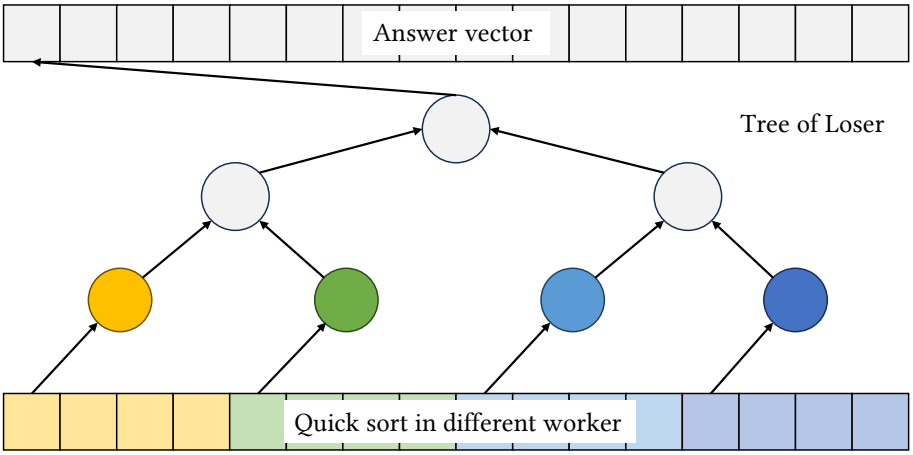

Figure 4: Tree of loser for k-way merge problem

Since we use workers to the power of 2, we can use a special data structure called tree of loser. This is a perfect binary tree, finding the minimum value only requires $\log_2 K$ comparisons, where K represents the number of workers.

When K is not to the power of 2, we can always complete to the power of 2 in the tree of loser. But a more easily implemented operation is to use the minimum heap, which is possible for any K.

2.2.1 Sub experiment: Tree of loser v.s. Heap

There is almost no difference in time between heap and tree of loser when workers = 32, and the time fluctuation should come from communication, so the experimental results will not be listed here

2.3 Profiling results

We can observe that as workers increase, cache misses and page faults decrease.

Number of cores	Average CPU cycles	Average cache misses	Average page faults
1	81,021,960,741	134,436,391	6,887
2	92,184,902,161	80,951,094	6,016
4	80,933,116,653	45,338,076	5,046
8	76,939,363,939	26,575,634	4,645
16	76,937,401,019	19,051,619	4,455
32	78,842,263,181	15,437,314	4,530

Table 3: Profiling results of task 1

3 Task 2: Process-Level Parallel Bucket Sort

3.1 Solution

According to the hints on the page of GitHub. We just split buckets into different groups and assign them to different workers to do the bucket sort. Each workers do a insertion sort for its buckets and send it back to Master node to merge them.

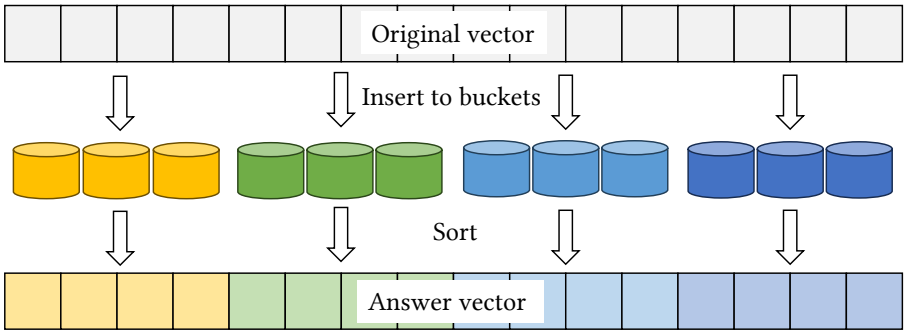

Figure 5: Bucket sort

3.1.1 Reason for using insertion sort

The best time complexity of insertion sort is $O(n)$ where n is size of the array. In insertion sort, if there are more elements that are already positioned correctly, the faster the running speed is. Because we partition according to the range of values of the elements, although the method of generating random numbers is a uniform distribution related to length, we still expect many already ordered ones in a bucket. However, other sorting algorithms do not accelerate due to their own good sorting like insert sorting.

3.1.2 Sub experiment: Can we use other sorting?

We have discussed the reason why we use insertion sort in bucket sort. But is there any other sorting algorithms to combining classic sorting algorithms?

The answer is yes. We can combine merge sort and insertion sort. This sorting algorithm is also called Tim Sort[2].

Workers	BucketSort(Insertion Sort)	BucketSort(Tim Sort)
1	11892	10838
2	8427	8371
4	5113	5072
8	3586	3406
16	2701	2755
32	2667	2469

Table 4: Comparison of different sorting (n=1000000, number of buckets=n/100)

We set RUN=32 in the Tim Sort. Since we set number of buckets=n/100. So we expect there are 100 numbers in a bucket so our strategy of Tim Sort works. There is a bit speed up.

[2]Reference: https://en.wikipedia.org/wiki/Timsort

3.2 Experiment: Send v.s. Gather

Since we cannot know how many elements in a bucket, we have two strategy to collect them.

One is using `send` operation and we provide a buffer with sufficient length to `recv`. A trick is that we can add -1 at the end of our sending vector since there are no negative number in the original vector to indicate that this is the end of our sending vector.

Another approach is sending all the length of buckets to Master and use `gather` operation to collect them.

Workers	BucketSort(Insertion Sort + Send)	BucketSort(Insertion Sort + Gather)
1	12272	11892
2	9698	8427
4	5488	5113
8	3788	3586
16	3067	2701
32	2671	2667

Table 5: Comparison of different collecting methods (n=1000000, number of buckets=n/100)

There is a bit speed up.

3.3 Experiment: Number of buckets

Undoubtedly, the number of buckets has a significant impact on efficiency. I did a experiment among different buckets for Insertion Sort (Tim Sort is not encouraged in this assignment).

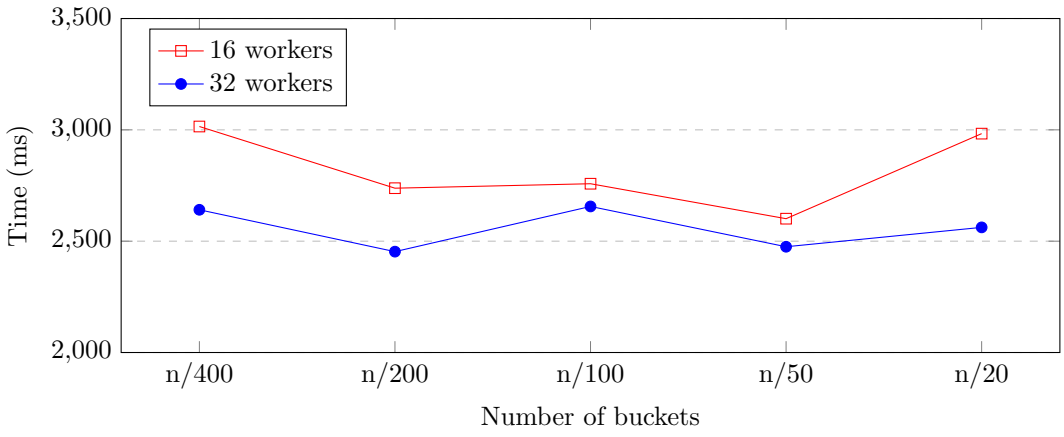

Figure 6: Compassion of execution time using different buckets (n=1000000)

So I choose n/50 as my number of buckets.

3.4 Profiling results

We can observe that as workers increase, cache misses and page faults decrease.

Number of cores	Average CPU cycles	Average cache misses	Average page faults
1	70,711,703,690	307,333,040	277,702
2	77,972,770,821	170,294,923	135,340
4	67,761,910,894	89,922,385	75,360
8	63,505,625,441	59,323,076	41,315
16	61,913,097,432	44,839,847	23,970
32	62,143,993,982	38,066,331	12,622

Table 6: Profiling results of task 2

4 Task 3: Process-Level Parallel Odd-Even Sort

4.1 Solution

According to hints, we split the vector into different workers. Each worker is responsible for one interval and it may communicate to its neighbors.

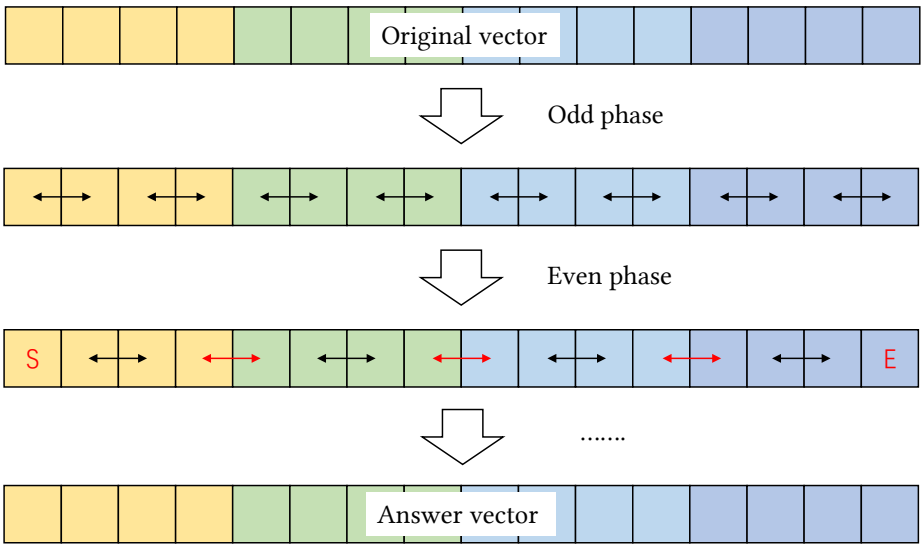

Figure 7: Odd-Even sort

The stop condition is every worker find their interval is sorted. So every cycle (one odd and one even phase), we use `MPI_Allreduce` to check if it is sorted.

4.2 Experiment: Do we really need to reduce every phase?

It is easy to see that `MPI_Allreduce` is a time-consuming operation. So I come up with the idea, we do not do the operation every cycle. We just do it after some cycle each time. I called it reduce cycle. So I do the experiment.

Cores	1	100	1000	10000
1	38797	39602	38794	38922
2	28951	29629	28698	28786
4	19113	18576	18279	18264

Cores	1	100	1000	10000
8	11885	10821	10669	10680
16	7733	5770	5674	5673
32	6355	2939	2804	2804

Figure 8: Compassion of execution time with different reduce cycle

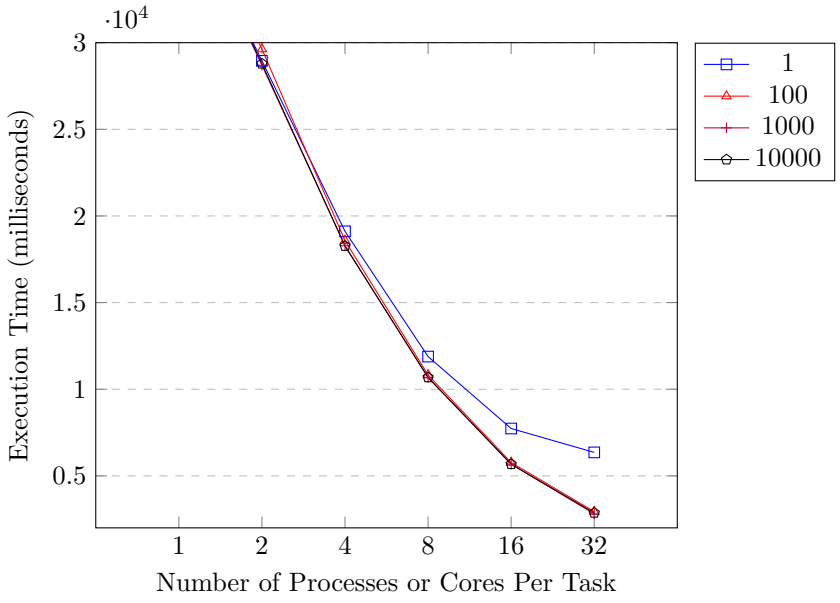

Figure 9: Execution time in Part B

We can see that the effect is significant, but there hasn't been much change since 1000. So I ended up taking 1000.

4.3 Profiling results

We can observe that as workers increase, cpu-cycles and page faults decrease.

Number of cores	Average CPU cycles	Average cache misses	Average page faults
1	112,279,708,993	150,281	3,434
2	82,889,874,496	53,047	2,809
4	52,169,816,501	2,389,668	2,449
8	30,354,507,742	3,828,971	2,384
16	16,147,766,938	4,240,322	2,301
32	8,054,338,142	3,223,692	2,199

Table 7: Profiling results of task 3

5 Task 4: Dynamic Thread-Level Parallel Merge Sort

5.1 Solution 1: OpenMP

The problem with OpenMP is that it creates many threads when entering the thread section and destroys them when exiting. If we recursively perform merge sort in this way, there is no doubt that there will be a lot of wasted time.

But if we use `#pragma omp parallel for`, OpenMP will handle it to avoid time wasting.

So a natural idea is that we don't recursively do merge sorting, we can use a loop to write merge sorting, which is very easy to implement because only one `#pagram` tag needs to be added.

```
int n = vec.size();
int gap = 1;
int* tmp = (int*)malloc(sizeof(int) * n);
```

```
while (gap<n) {
    #pragma omp parallel for
    for (int i = 0; i < n; i += (gap << 1)) {
        int j = i;
        int begin1 = i, end1 = begin1 + gap - 1;
        int begin2 = begin1 + gap, end2 = end1 + gap;
        if (end1 >=n || begin2 >=n) continue;
        if (end2 >=n) end2 = n - 1;
        my_merge_normal(vec, begin1, end1, end2);
    }
    gap <<= 1;
}
free(tmp);
```

5.2 Solution 2: std

After C++11, we not only have the threads class, but also the future class, which we can asynchronously call.

At the same time, we cannot create too many threads, so we set a threshold. If the length is below this threshold, we do it in sequential way. Above this threshold, both recursion and merging will create new threads.

```
if (right - left < MIN_PARALLEL_SIZE) {
    mergeSort(input, left, mid);
    mergeSort(input, mid + 1, right);
    my_merge_normal(input, left, mid, right);
} else {
    auto fut = std::async(std::launch::async, mergeSort, std::ref(input), left, mid);
    mergeSort(input, mid + 1, right);
    fut.wait();
    std::pair<int,int> p = findMedianSortedArrays(input,left,mid,right);
    auto futt = std::async(std::launch::async,
                           my_merge,
                           std::ref(input),
                           left, p.first,
                           mid + 1, p.second,
                           left);
    my_merge(input,
             p.first + 1, mid,
             p.second + 1, right,
             p.first + p.second - mid + 1);
    futt.wait();
    for(int k = left; k <= right; ++k) input[k] = temp[k];
}
```

5.3 Experimental results

Workers	OpenMP+Median	std+Median
1	12157	12780
2	12153	12760
4	6796	6663
8	4130	3753
16	2867	2361
32	2694	1726

Table 8: Comparison of different methods

5.4 Profiling results

Number of cores	CPU cycles	cache misses	page faults
1	114,822,867,691	350,215,525	686,191
2	114,777,243,876	350,996,499	686,350
4	114,403,488,283	348,562,579	686,728
8	113,911,843,266	347,474,173	1,108,833
16	113,520,760,711	345,505,108	686,451
32	118,142,474,111	350,593,604	685,640

Table 9: Profiling results of task 5

6 Task 5: Dynamic Thread-Level Parallel Quick Sort

6.1 Solution 1: std

Same as task 4, I use std and future library to asynchronously call.

Similarly, we set a threshold. If the length is below this threshold, we do it in sequential way. Above this threshold, we create new threads.

```
if (left < right) {
    int pivotIndex = partition(input, left, right);

    if (right - left < MIN_PARALLEL_SIZE) {
        quickSort(input, left, pivotIndex - 1);
        quickSort(input, pivotIndex + 1, right);
    } else {
        auto fut = std::async(std::launch::async,
        quickSort, std::ref(input), left, pivotIndex - 1);
        quickSort(input, pivotIndex + 1, right);
        fut.wait();
    }
}
```

6.2 Solution 2: thread pool

I tried to manage a simple thread pool to handle the input. But the performance is not good. The reason is that I implemented a queue with a lock, and if I enqueue the queue many times, I will get a poor performance.

Because I didn't have enough time, I didn't implement complex thread pools, such as lockless queues and thread theft. The code can be referenced `src/quicksort/parallel_trread_pool.cpp`

6.3 Profiling results

Number of cores	CPU cycles	cache misses	page faults
1	97,939,541,748	184,779,904	5,591
2	97,933,372,026	183,093,595	5,512
4	97,409,868,611	182,897,839	5,286
8	96,810,713,683	184,200,987	5,124
16	95,953,519,320	180,656,629	5,198
32	91,878,558,563	170,017,065	6,876

Table 10: Profiling results of task 5