Independent Study

Final Report

Computer Science Department

Kean University

Student: Chen-Kai Tsai

Instructor: Dr. Jing-Chiou Liou

Date: April 18th, 2020

Table of Content

1. Introduction

We are now in the programming environment where almost all the devices have multicore processors. From INTEL® CORE™ i7 PROCESSORS in your personal computer to Apple A13 Bionic in your iPhone, they are all multicore processors. However, your program will not automatically use multiple cores when they are running on those processors. It depends on how you write your code to take advantage of this ability. How to exploit multicore processor to make our program run faster? We need parallel computing and parallel algorithms to parallelize our software. There are many different parallel algorithms however, in our study, we focusing on applying sorting algorithms on parallel programming and analyze their behavior [1][2][3][12]. We select six sorting algorithms including Shell Sort, Bitonic Sort, Merge Sort, Rank Sort, Selection Sort and Odd-Even Sort which are parallelized and tested under different scenarios to observe their behavior. The first two set of tests are strong scaling performance tests and weak scaling performance tests with pure integer datasets. The third set of tests are testing the behavior of the algorithm under uniform distribution datasets with different degree of duplication. The final set of tests are applying and modifying these parallel sorting to sort real-world data in databases. In this research, we provide the result of the tests and analysis of the reason behind their behavior either theoretically or with implementation aspect. This research aims to help programmer understand what need to be taken into consideration when choosing sorting algorithms and serve as a basis for their further analysis.

2. Implicit Threading Library and OpenMP

Implicit threading libraries take care most of the detail for creating, managing and synchronizing threads. With this feature, programmer can focus on creating the parallel algorithms and the compiler will generate codes for creating and managing threads. However, the bad side of this feature is that the flexibility of the parallel programs will be limited by what setting options for thread managing these libraries provide. OpenMP is an application programming interface for share memory multiprocessing programming in C, C++ and Fortran. The API provide compiler directives, runtime functions and environment variables for programmers to create, synchronize and manage threads with fork-join model. The master thread will fork off threads in the parallel region and join those threads at the end of the parallel region. Since programming only need to insert compiler directives into their codes, the parallel version of a program can be created with minimum modifications to the original sequential one. Codes write with OpenMP are portable among different system environments. Even if compilers do not support it and ignore compiler directives, the code still can be run on that system without parallelization.[4][11]

3. Test Environment and Test datasets

The following test is performed on a system with following components

1) Operating system: Ubuntu 18.04.4 LTS with POSIX thread model
2) CPU: Intel Core i7-9750H with 6 cores, 12 threads and frequency from 2.6 GHz to 4.5 GHz.
3) Memory size: 32GB
4) GCC version: 7.4.0 with OpenMP version 4.5
5) MariaDB version 10.1.44

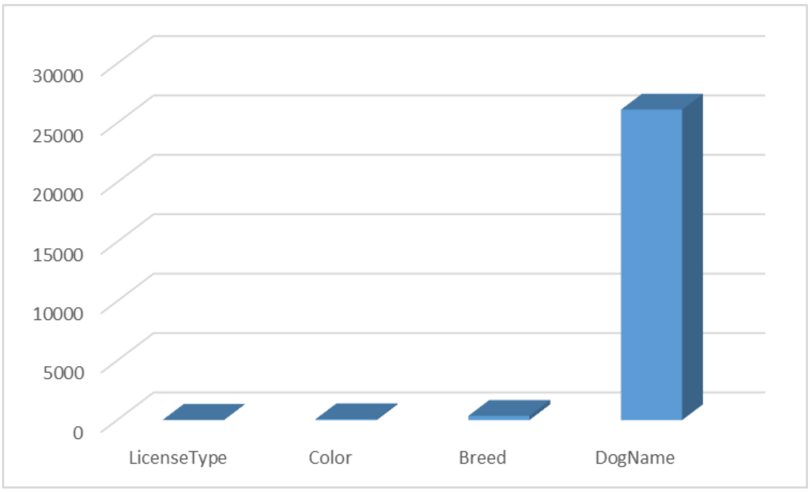
The first section of the result is the strong scaling performance test and the weak scaling performance test of the six selected sorting algorithms. The details of the test datasets are as following:

Sorting algorithms in both strong scaling test and weak scaling test are divided into two groups. The first group including Shell Sort, Merge Sort and Bitonic Sort, generally have better performance and need larger dataset to better show their scaling performance. On the other hand, the second group including Rank Sort, Selection Sort and Odd-Even Sort, have worse performance and the dataset for the first group will take too much time to execute.

The first group of sorting algorithms are tested on a integer dataset with 2GB (the number of integers is 1024*1024*512) data size that is generated by the rand() specified in C stdlib.h. The rand() function will use linear feedback shift register algorithm mentioned in the ISO C standard to generate the dataset. The second group of sorting algorithms are tested on an integer dataset generated by the same generator but with smaller 1MB data size (the number of integers is 1024*256). These two tests are testing the strong scaling performance of sorting algorithms with same size of dataset and increasing number of threads. The third and fourth tests are weak scaling performance tests. The test dataset is generated by the same generator but for each round the size of the dataset and the number of threads is doubled. The dataset size start from 256MB for the third test including first group of sorting algorithms and the dataset size for the fourth test including second group of sorting algorithms start from 128KB.

The second section of the result is testing “How the amount of duplicated number in the test dataset will affect the performance of the sorting algorithms?”. The dataset is generated by generator defined in C++ STL with predefined random number generator “default_random_engine” and random number distributions “uniform_int_distribution”. The dataset content evenly distributed numbers with a defined range. For Shell Sort, Merge Sort and Bitonic Sort, there are four datasets with the same data size which is 2GB. The first dataset content evenly distributed numbers that for each value, there will be 1024 elements in the data set so totally there are 1024*512 unique elements while the second dataset has 1024*1024*4 unique elements. The third dataset and fourth dataset are 1024*1024*32 and 1024*1024*256 unique elements. For Rank Sort, Selection Sort, Odd-Even Sort, the dataset is generated with same generator and as the previous set numbers of unique elements are increased by 8 times for each dataset.

The final section of the result is applying those sorting algorithms to sort data in the database. The testing data is public dataset published by Allegheny County / City of Pittsburgh / Western PA Regional Data Center and maintained by Samuel Mazza (Samuel.Mazza@alleghenycounty.us). We include fields “LicenseType”, “Breed”, “Color”, “DogName” in our database sorting. The following diagram shows the amount of unique data in the dataset.



4. Sorting Algorithms

All the sorting algorithm are implemented with pure C language style code and OpenMP for parallelization without using any C++ language features and C++ standard library.

4.1 Shell Sort

Shell Sort is based on the concept of Insertion Sort and add a “gap” to make the sorting process faster. The first version of Shell Sort was published by Donald Shell in 1959 [6]. Instead of comparing and moving the element one position at a time, Shell Sort use a gap to jump and compare the element that is a gap away. The value of the gap will decrease until it becomes 1 and at this stage Shell Sort is equal to an Insertion Sort. There are a lot of ways to generate the gap sequence and we implement the gap sequence introduce by Knuth, Donald E. in 1997 [5] with the following C code.

```
// “h” is the gap and “N” is the size of the dataset.
h = 1;
while (h < N)
      h = 3 * h + 1;
```

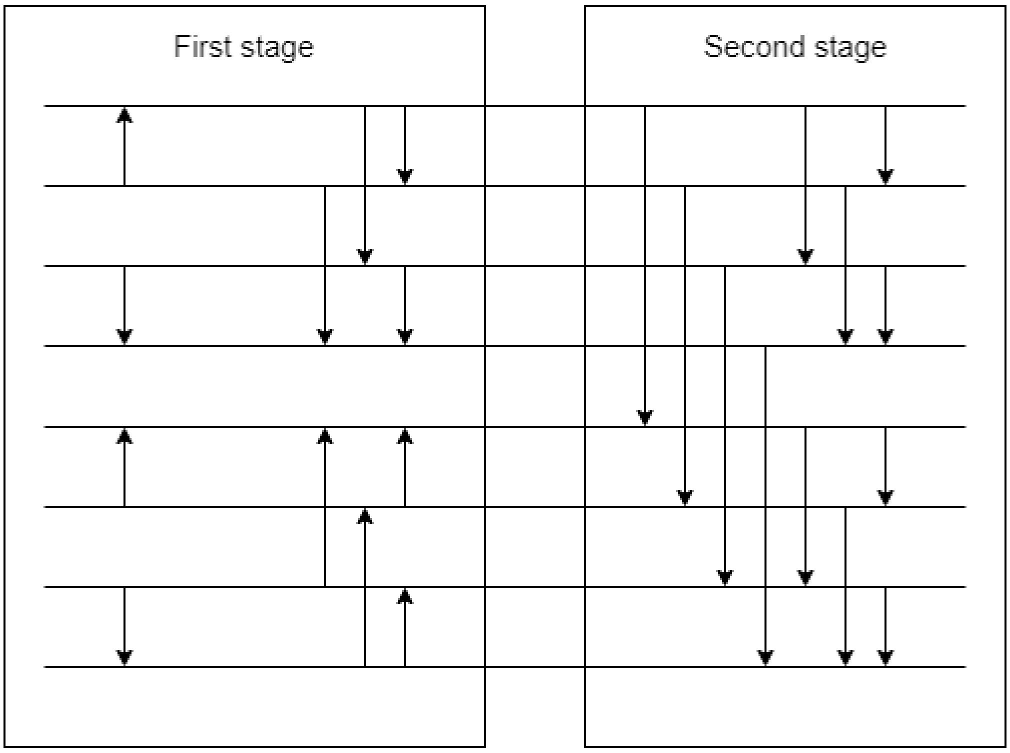
4.2 Bitonic Sort

Bitonic Sort which was designed by Kenneth Edward Batcher [7] is a sorting network with predefined directions and sequence. A bitonic sequence is defined as following:

A sequence of numbers with n elements is bitonic if there is an index i in the sequence that:

$$x_0 \le x_1 \le x_2 \le \cdots \le x_i \; and \; x_i \ge x_{i+1} \ge x_{i+2} \ge x_n$$

A Bitonic Sort contain two stages. The first stage is converting an unsorted dataset into a bitonic sequence and the second stage is sorting a bitonic sequence.



An example of Bitonic Sort with 8 elements

4.3 Merge Sort

Merge Sort splits the sequence into multiple subsequences until there is only one element in the sequence and then merge them into a sorted sequence. In our implementation, the number of the subsequences that the Merge Sort divides is depended on the number of threads that set to the parallel program. Each thread will sort their own subsequence by sequential version of the Merge Sort. In order to avoid bad load balance, when the number of threads assigned that is bigger than the nearest power of two value will be ignored. For example, if the number of threads is assigned to three, the program will only use two threads.

4.4 Rank Sort

Rank Sort is simply traversing through the entire sequence for every element and count the number of elements that is bigger than the selected element. After ranking an element, store the element to the other storage to the position which is determined by its rank. The implementation of the parallel version is done by distributing elements in the sequence to every thread. Since there is no data dependency, this is an embarrassingly parallel problem theoretically. However, the false sharing will occur when storing the ranked element to the new storage. As the number of threads increase the probability of false sharing will increase.

4.5 Selection Sort

Selection Sort is a simple algorithm that for every element select the biggest or smallest element in an unsorted part and then move it to the sorted part of the sequence. The parallel version of Selection Sort involving a reducing phase that the local maximum will have to compare with each other to find the biggest element. OpenMP provide a clause for built-in algorithm for reduction.

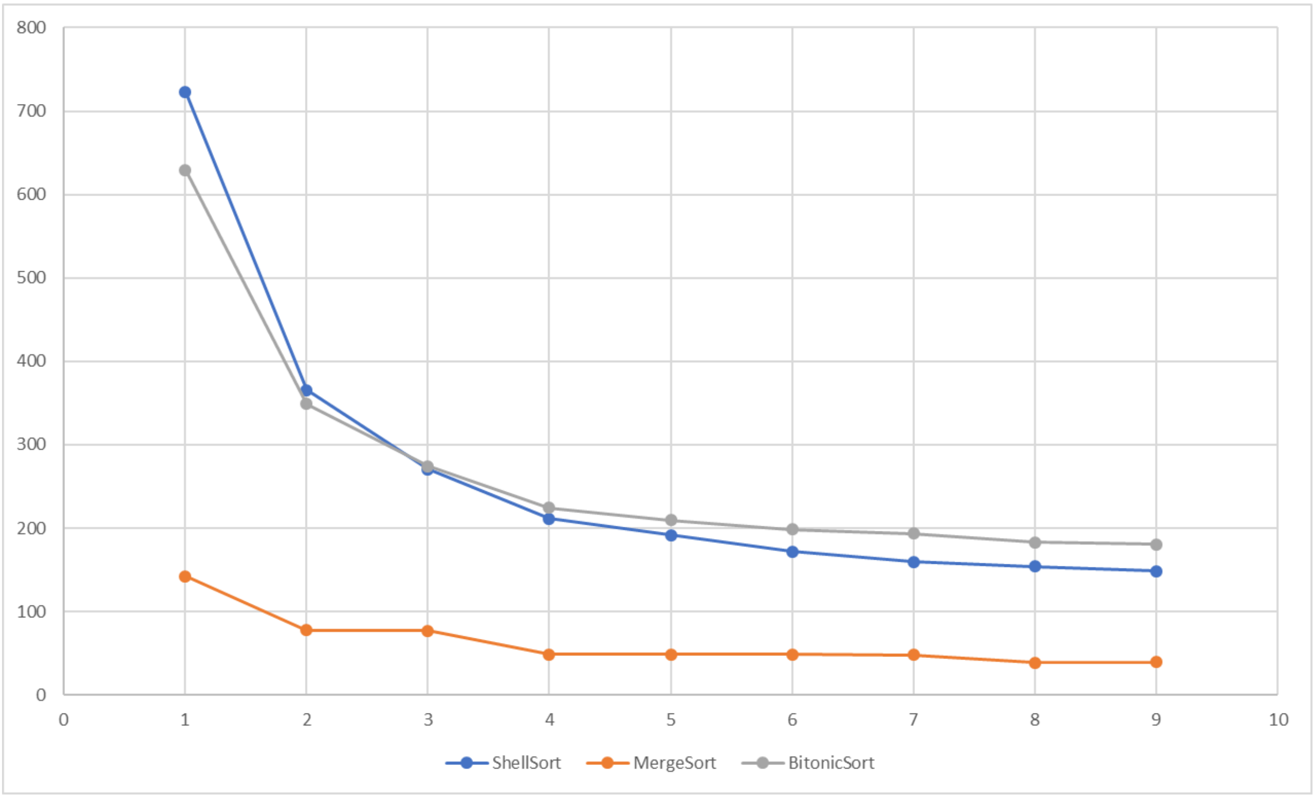
4.6 Odd-Even Sort

Odd-Even Sort is a parallel version of Bubble Sort and has two stages which are odd stage and even stage. At odd stage, the element with odd index will compare and swap with the adjacent even-index element while at even stage, the element with even index will do the same thing to the adjacent odd-index element. These two stages will repeat until the sequence is sorted.

5 Results

Diagrams below are the results of different tests. For each diagram, there is a paragraph which explain the result and the reason behind it with the C code that implement these algorithms.

5.1 Strong Scaling Performance Test for Shell Sort, Merge Sort and Bitonic Sort

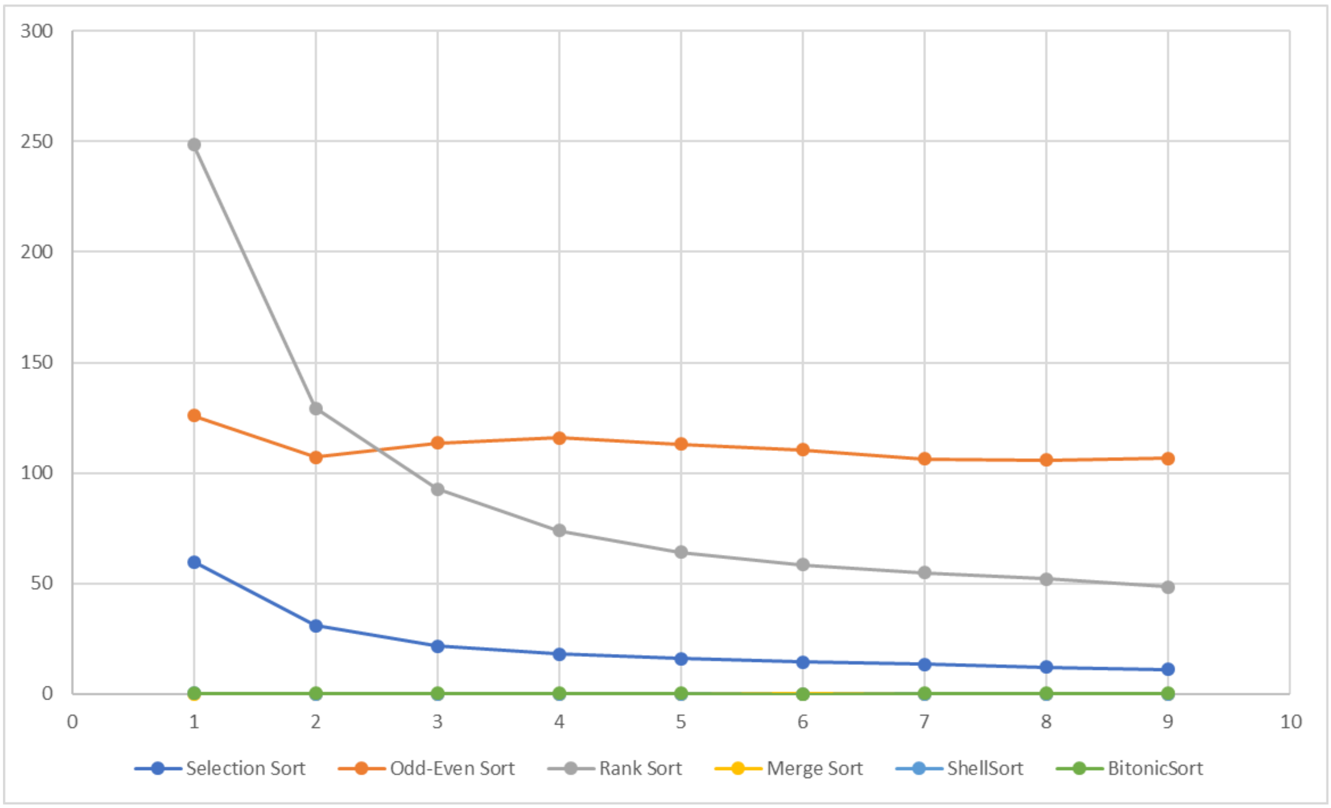


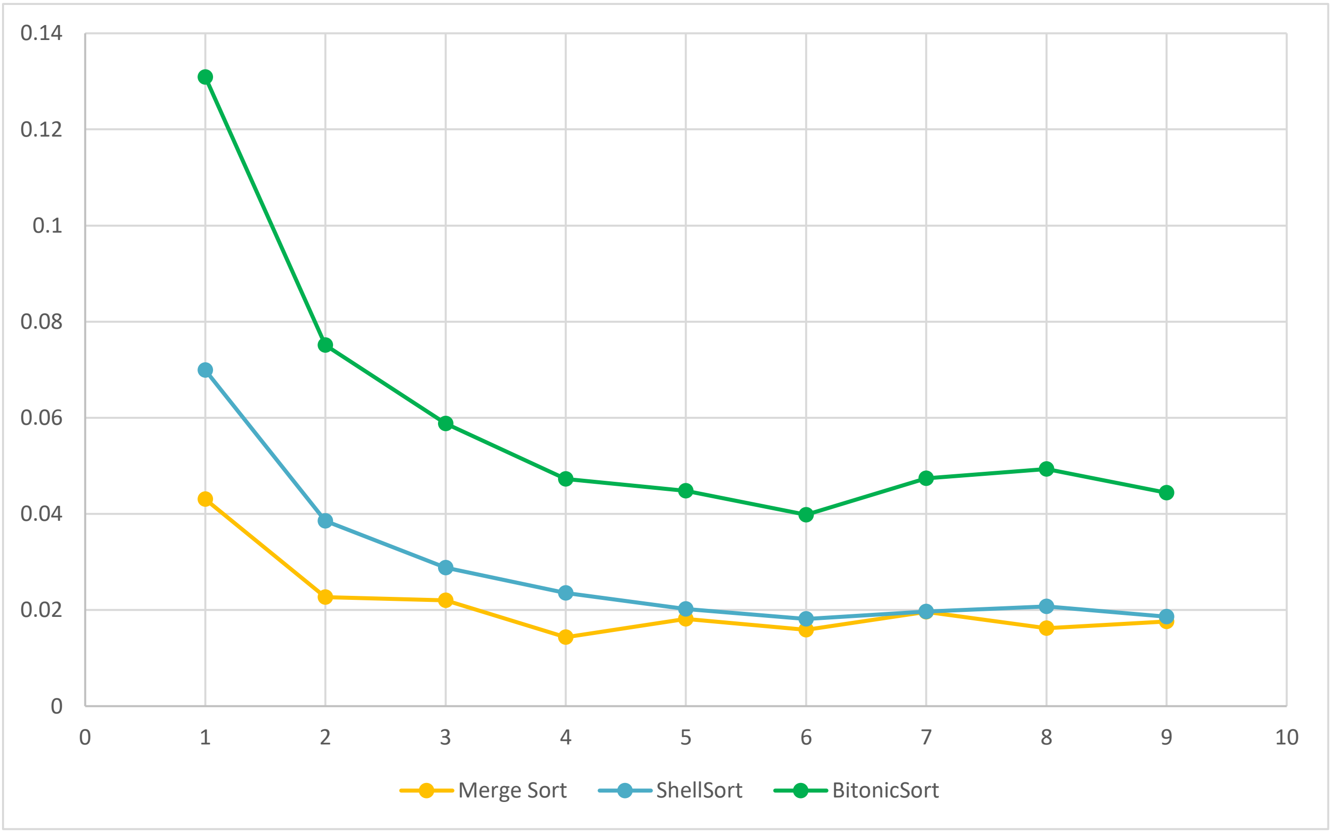
The x axis is the number of threads that the parallel program uses and the y axis is the execute time in seconds. The data size of this test is 2GB (1024*1024*512 elements)

Merge Sort has generally better performance among these three because it can be naturally adopted to parallel simply by distribute portion of the dataset to other threads. Although this test is conducted under a multi-core CPU, the number of cores is small and the test result is heavily affected by the sequential performance. Both Bitonic Sort and Shell Sort are not work optimal and their time complexity are

$O(nlog^2(n))$ and $O(n^{\frac{3}{2}})$ (depend on the chosen gap) respectively. As for Merge Sort, its complexity is $O(n(logn))$ which has best sequential performance.

5.2 Strong Scaling Performance test for Rank Sort, Selection Sort, Odd-Even Sort, Shell Sort, Bitonic Sort and Merge Sort





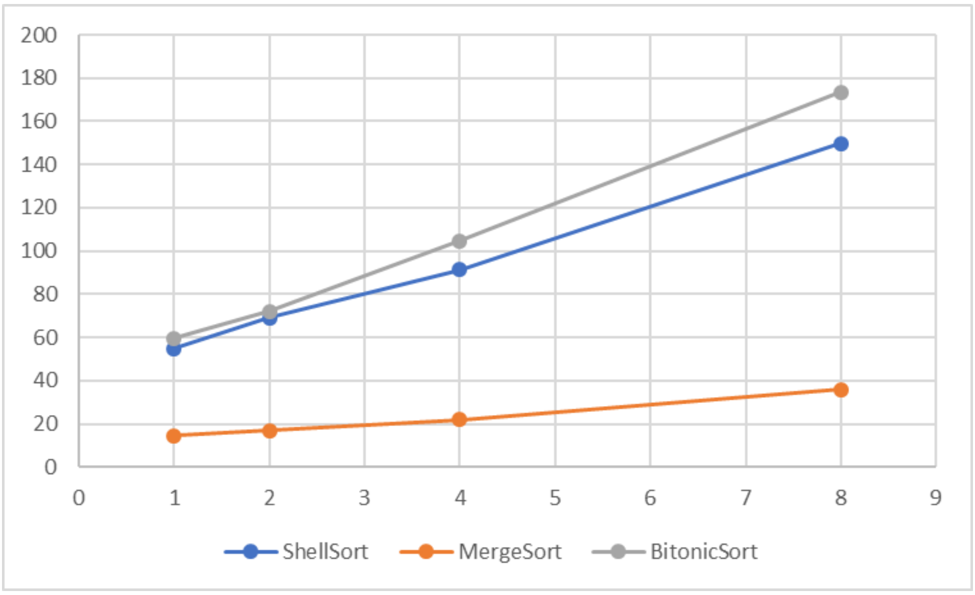
The x axis is the number of threads that the parallel program uses and the y axis is the execute time in seconds. The data size of this test is 1MB (1024*256 elements)

The reason behind the slow performance of the sorting algorithm in second group is the same reason that explained in 4.1. Rank Sort, Selection Sort and Odd-Even Sort have same sequential complexity $O(n^2)$. In the first diagram, the Odd-Even Sort did not have good scaling performance. It is because the implementation of the Odd-Even Sort spends most of the time creating and destroying the threads. The following C code is presented for better explanation.

```c
while (exch)
{
    exch = 0;
    #pragma omp parallel num_threads(NTHREAD)
    {
        int temp;
        #pragma omp for
        for (i = 0; i < N - 1; i += 2)
        {
            if (input_data[i] > input_data[i + 1])
            {
                temp = input_data[i];
                input_data[i] = input_data[i + 1];
                input_data[i + 1] = temp;
                exch = 1;
            }
        }

        #pragma omp for
        for (i = 1; i < N - 1; i += 2)
        {
            if (input_data[i] > input_data[i + 1])
            {
                temp = input_data[i];
                input_data[i] = input_data[i + 1];
                input_data[i + 1] = temp;
                exch = 1;
            }
        }
    }
}
```

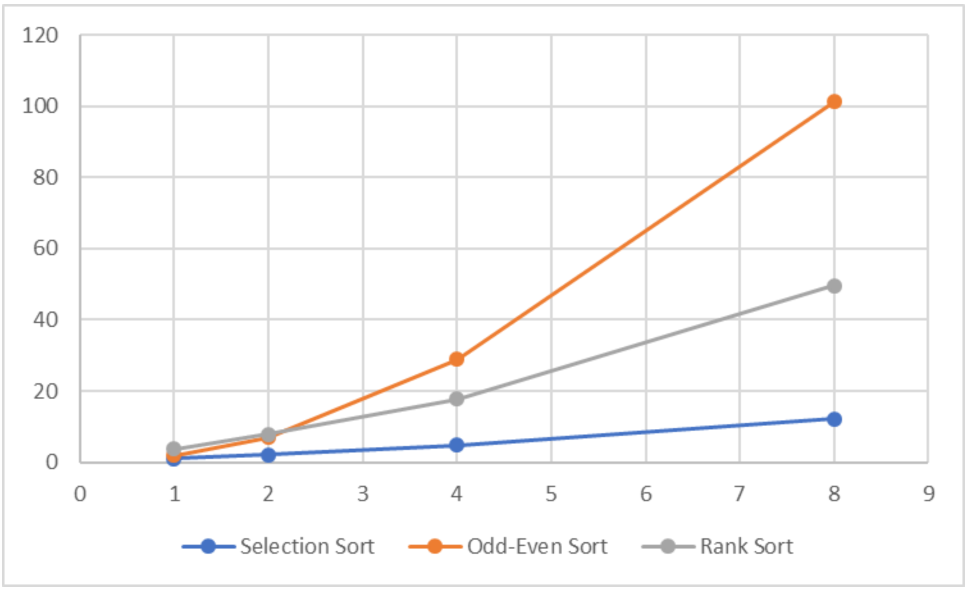
Threads will be created at `#pragma omp parallel num_threads(NTHREAD)` and stay alive until the end of the parallel region (line 30) but, the parallel region is inside a while loop. It means that the program will constantly create and destroy threads for each while loop.

5.3 Weak Scaling Test for Shell Sort, Bitonic Sort, Merge Sort



The x axis is the number of threads is equal to and the multiplier for dataset size (Starting from 256MB). The y axis is the execute time in seconds.

5.4 Weak Scaling Performance Test for Selection Sort, Odd-Even Sort and Rank Sort



The x axis is the number of threads is equal to and the multiplier for dataset size (Starting from 128KB). The y axis is the execute time in seconds.

The results of weak scaling performance test are conducted by doubling the number of cores and the data size at each run. The ideal result of weak scaling performance test is a flat line which indicated that the execution times are a constant. However, in reality, algorithms take more execution time to execute when the data size is increasing. The explanation will be made by using Selection Sort as the example.

The time complexity of Selection Sort is $O(n^2)$ and ideally the parallel version of complexity can be estimate by dividing the number of threads T. By doubling n and T, the complexity is $\frac{O(4n^2)}{2T}$. Therefore, we can expect that the execution time will be two times longer. The weak scaling performance testing results for Selection Sort are as following:

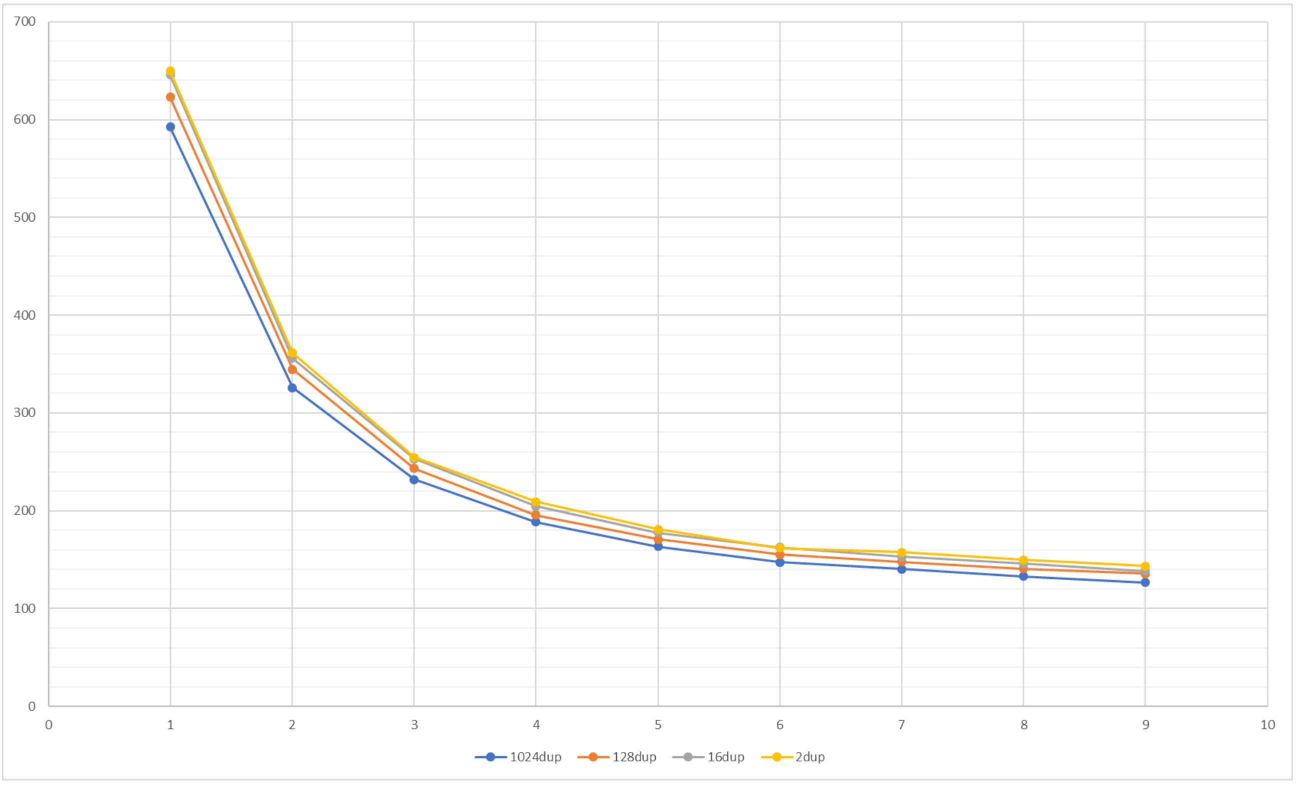
Cores	Execution Time
1	1.01448
2	2.11315
4	4.71368
8	12.1462

At each run of test, the Execution Time in each run is approximately two times longer than the previous run (The overhead of multi-threading needs to be put into consideration so the increasing of execution time is more than two times), thus match the expectation of the theoretically analysis.

5.5 Uniform Dataset Tests

Using uniform dataset with different degree of duplication for these tests is aim to find out whether the amount of duplicated data in the dataset will pose effect on the performance of sorting algorithms.

- Shell Sort

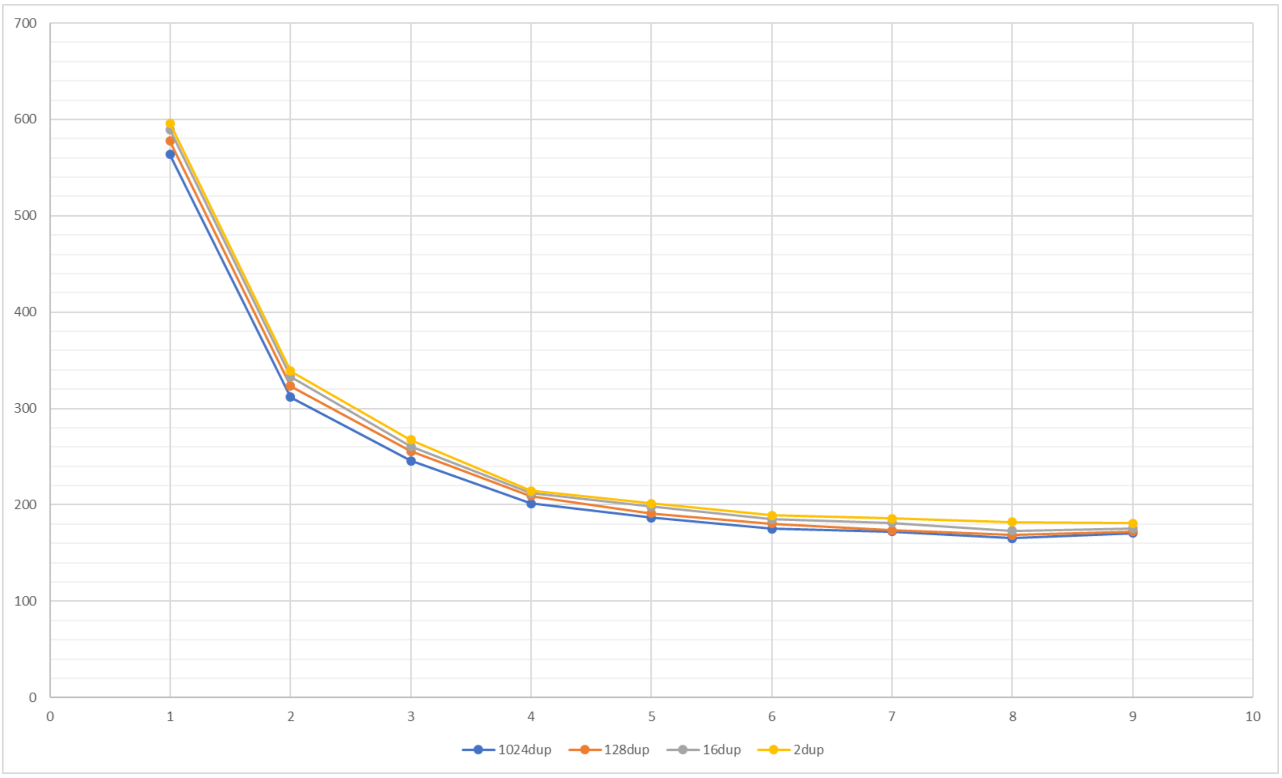


The x axis is the number of threads that the parallel program uses and the y axis is the execute time in seconds. The data size of this test is 2GB (1024*1024*512 elements). Each line indicates different distributions of the testing dataset.

```c
h = 1;
while (h < N)
    h = 3 * h + 1;
h /= 3;

#pragma omp parallel firstprivate(h) num_threads(NTHREAD)
{
    while (h != 1)
    {
        #pragma omp for
        for (k = 0; k < h; k++)
        {
            int i, j, v;

            for (i = k; i < N; i += h)
            {
                v = input_data[i];
                j = i;
                while (true)
                {
                    if (j - h < 0)
                        break;
                    if (input_data[j - h] > v)
                    {
                        input_data[j] = input_data[j - h];
                        j -= h;
                        if (j <= h)
                            break;
                    }
                    else
                        break;
                }
                input_data[j] = v;
            }
        }
        h /= 2;
    }
}
```

Shell Sort is based on comparing and swapping. Additionally, a big number is gradually pushed to the top as the value of gap reduces. The time difference is caused by numbers with same value that will not do the swap. Therefore, the execute time is smaller when the amount of duplicated data is bigger.

- Bitonic Sort

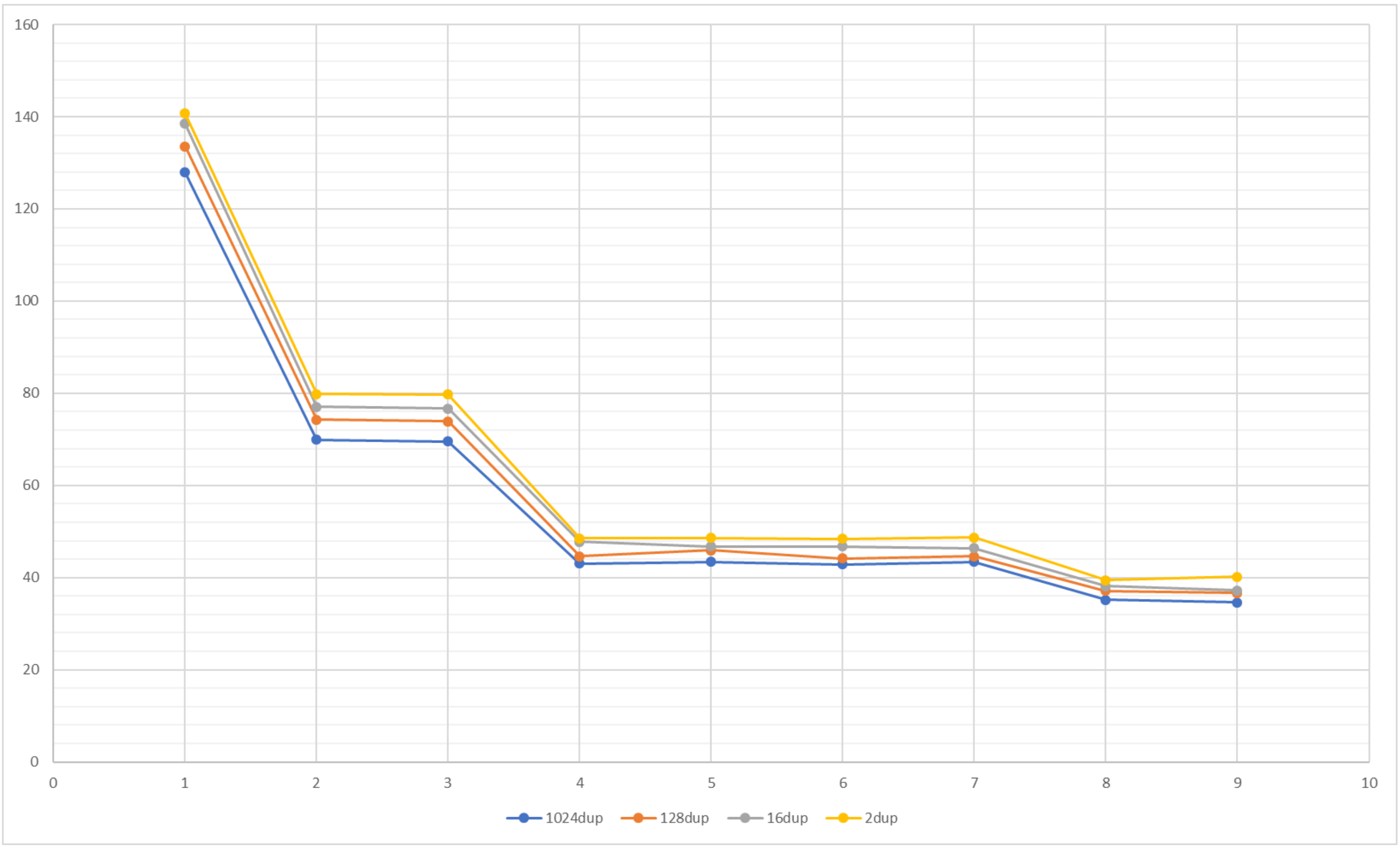


The x axis is the number of threads that the parallel program uses and the y axis is the execute time in seconds. The data size of this test is 2GB (1024*1024*512 elements). Each line indicates different distributions of the testing dataset.

```
void bitonicMerge(int start_index, int end_index, int dir, int *input_data)
{
    if (dir == 1)
    {
        int num_elements = end_index - start_index + 1, temp;
        for (int j = num_elements / 2; j > 0; j = j / 2)
        {
            for (int i = start_index; i + j <= end_index; i++)
            {
                if (input_data[i] > input_data[i + j])
                {
                    temp = input_data[i + j];
                    input_data[i + j] = input_data[i];
                    input_data[i] = temp;
                }
            }
        }
    }
    else
    {
        int num_elements = end_index - start_index + 1, temp;
        for (int j = num_elements / 2; j > 0; j = j / 2)
        {
            for (int i = start_index; i + j <= end_index; i++)
            {
                if (input_data[i + j] > input_data[i])
                {
                    temp = input_data[i + j];
                    input_data[i + j] = input_data[i];
                    input_data[i] = temp;
                }
            }
        }
    }
}
```

Bitonic Sort is also based on comparing and swapping so the reason and phenomenon is the same as Shell Sort. The amount of time swapping data is reduced due to the duplicated data in the dataset.

- Merge Sort

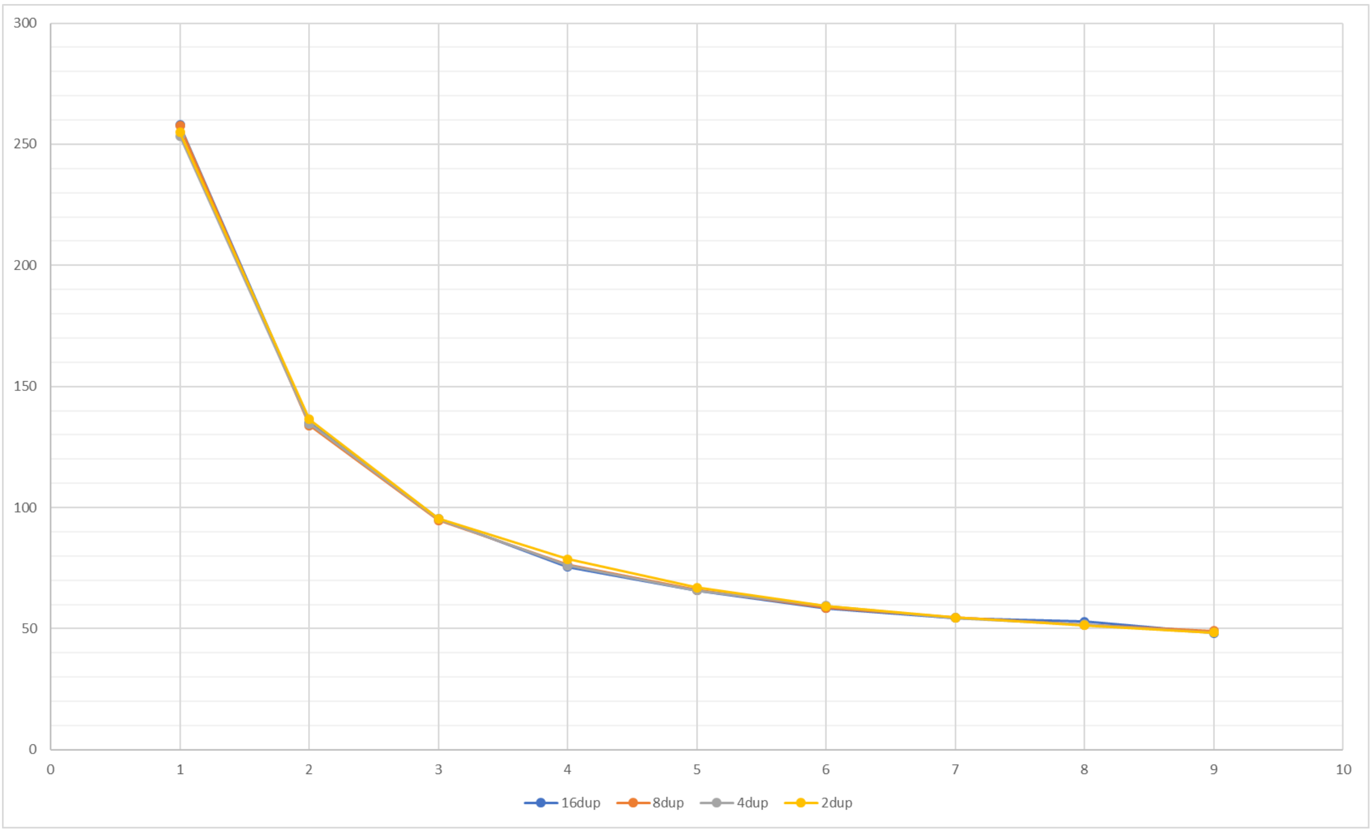


The x axis is the number of threads that the parallel program uses and the y axis is the execute time in seconds. The data size of this test is 2GB (1024*1024*512 elements). Each line indicates different distributions of the testing dataset.

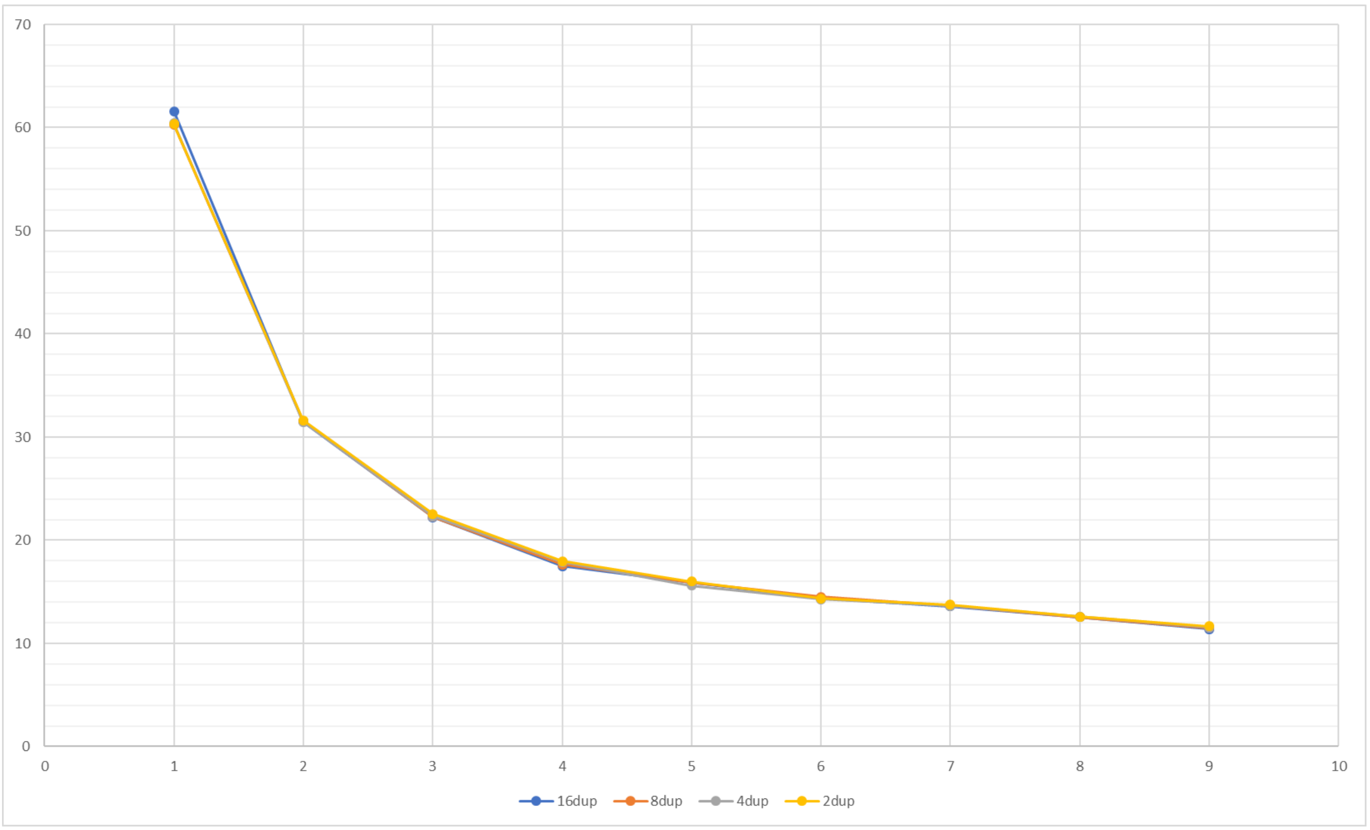
```c
void merge(int *input_data, int i1, int j1, int i2, int j2)
{
    int i, j, k;
    i = i1;
    j = i2;
    k = 0;

    int temp_size = j2 - i1 + 1;
    int *temp = (int *)malloc(temp_size * sizeof(int));

    while (i <= j1 && j <= j2)
    {
        if (input_data[i] < input_data[j])
            temp[k++] = input_data[i++];
        else
            temp[k++] = input_data[j++];
    }

    while (i <= j1)
        temp[k++] = input_data[i++];

    while (j <= j2)
        temp[k++] = input_data[j++];

    for (i = i1, j = 0; i <= j2; i++, j++)
        input_data[i] = temp[j];

    free(temp);
}
```

The merge phase of the merge sort will be affected by the number of duplicated elements in the dataset because at the merge part, there are two parts of the array need to be merged together. If a lot of duplicated elements from one part are put into the temp, that part of array will be consumed faster than the other one. Therefore, once one part of the array has all been put into the temp the other part of the array will be appended to the temp without doing any comparison and reduce execution time.

- Rank Sort



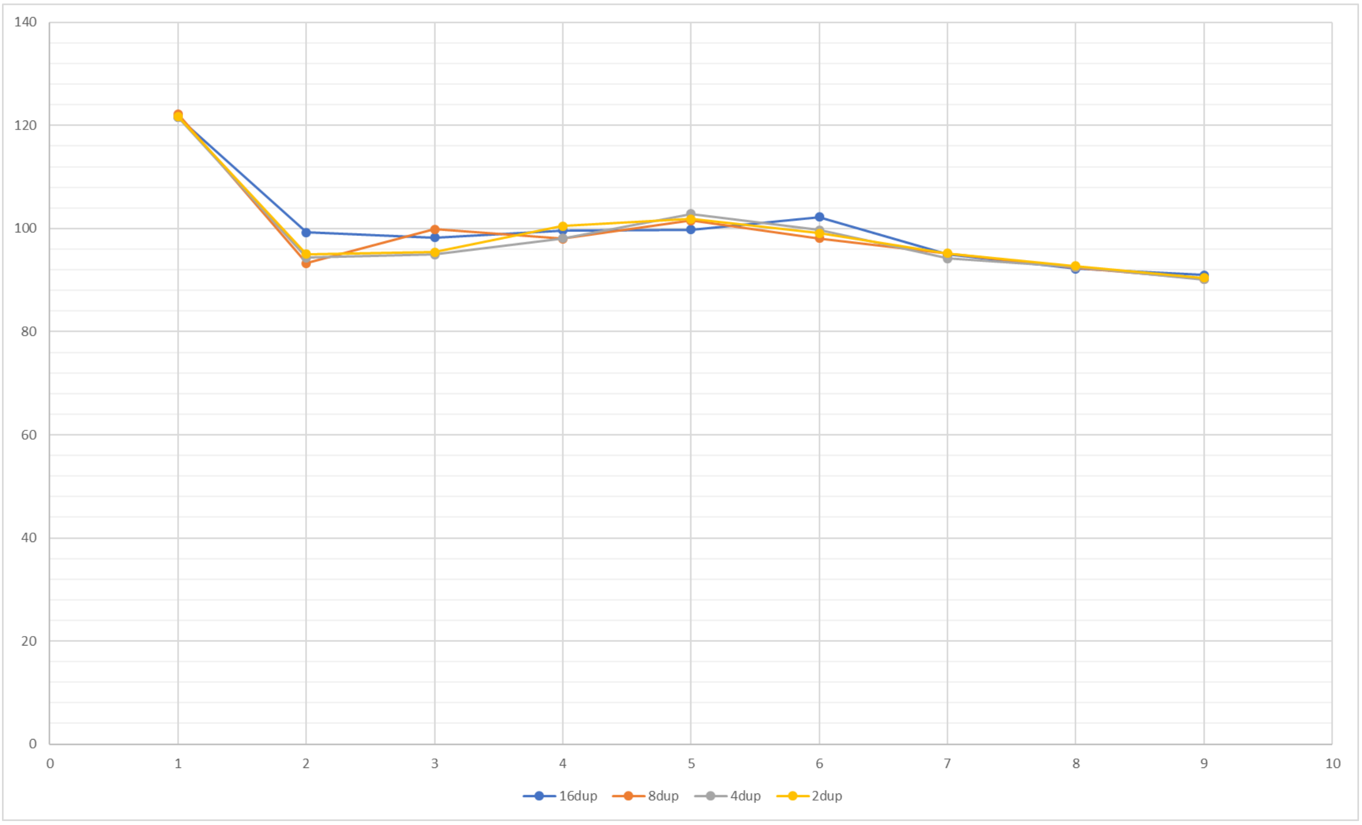
- Selection Sort



The x axis is the number of threads that the parallel program uses and the y axis is the execute time in seconds. The data size of this test is 1MB (1024*1024*512 elements). Each line indicates different distributions of the testing dataset.

Rank Sort and Selection Sort are not under the influence of the number of duplicated elements. For Rank Sort, it always traverses the entire array to count the rank and move the element to its final spot which means the element will not be moved again. For Selection Sort, for each loop it always finds the biggest element in the part which is not sorted yet and move it to the sorted part. For each element, there will be only one movement specifically done for it. To sum up, these two algorithms always do the same amount of instructions no matter what value of the elements in the dataset are.

- Odd-Even Sort

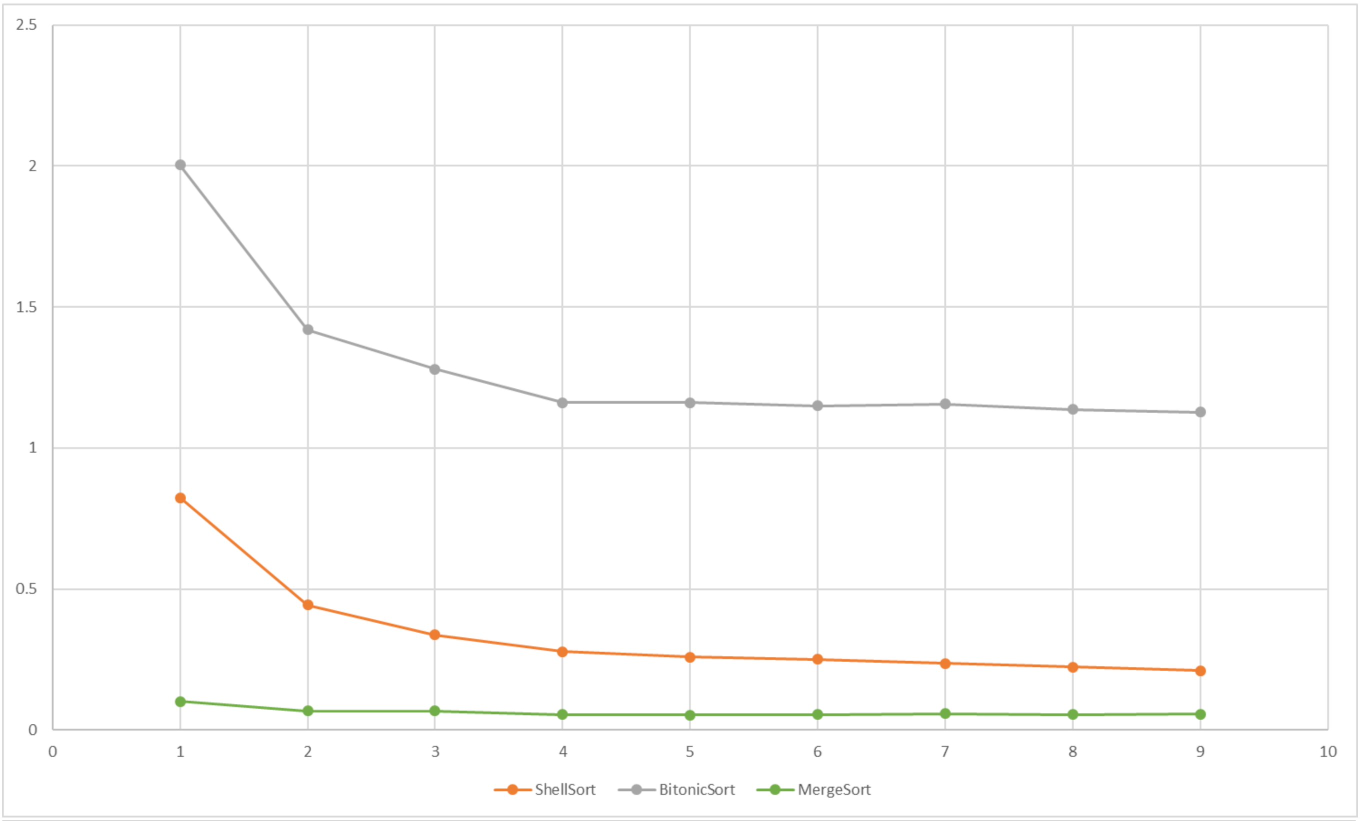


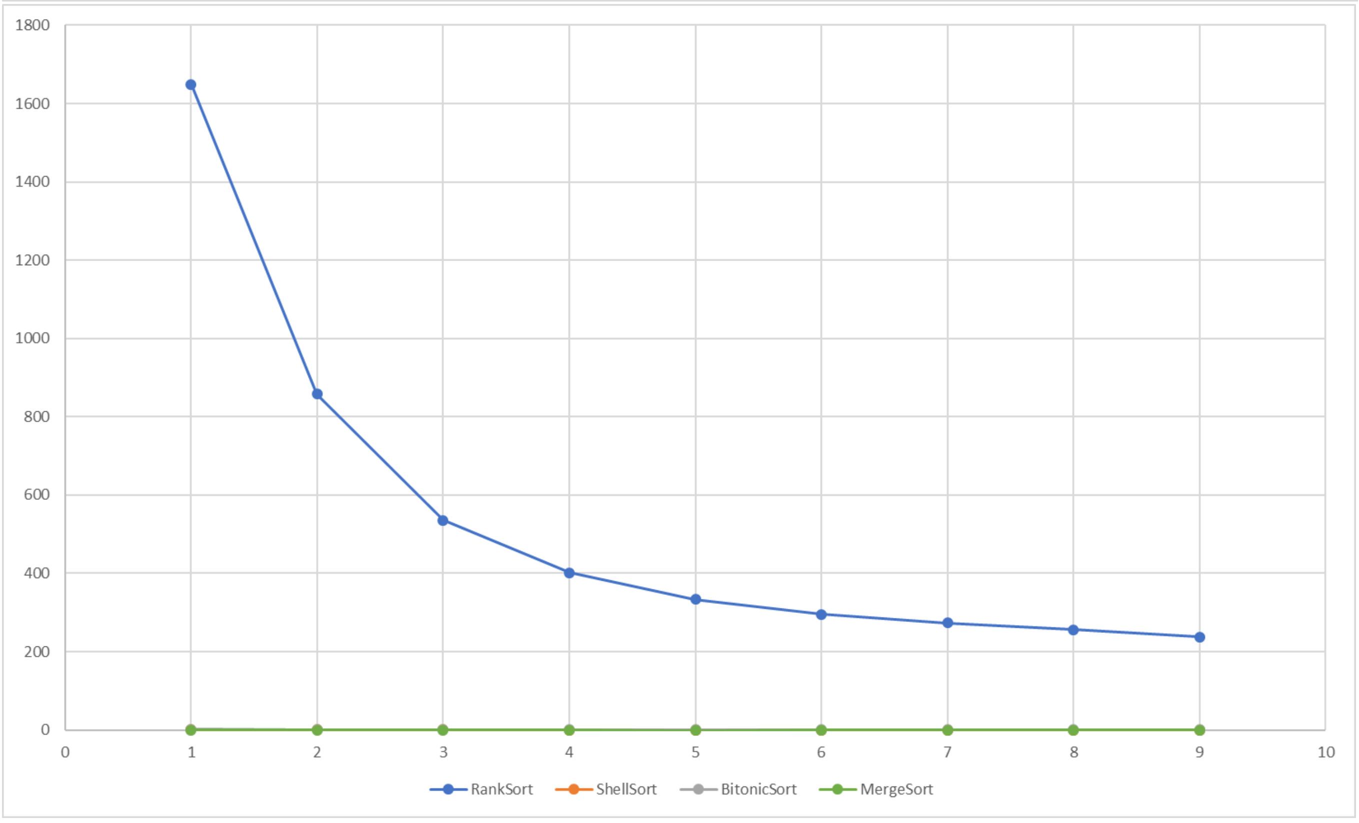
The x axis is the number of threads that the parallel program uses and the y axis is the execute time in seconds. The data size of this test is 1MB (1024*256 elements). Each line indicates different distributions of the testing dataset.

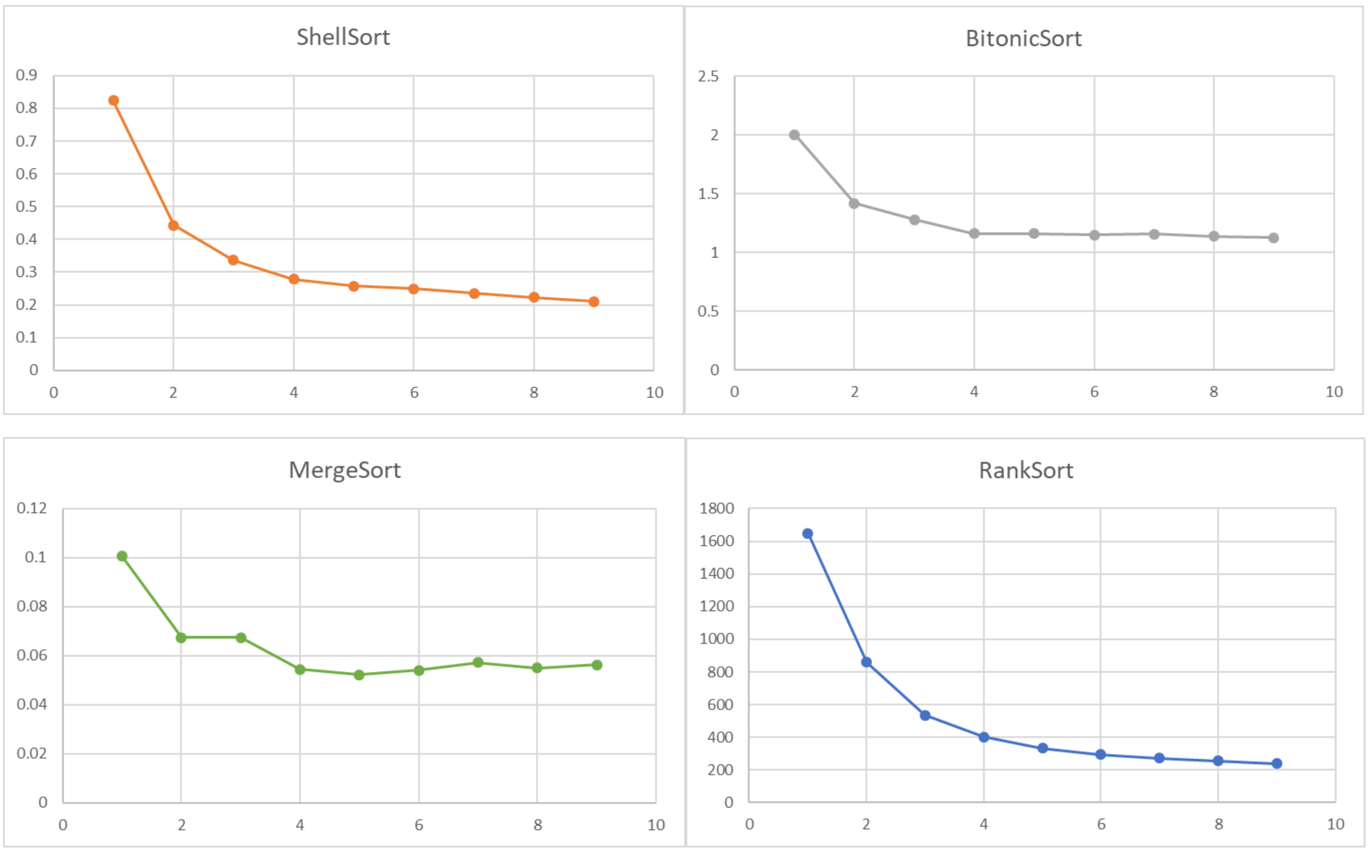
As describe in strong scaling test, this implementation spends most of its execution time on creating and destroying thread and the amount of computation for each thread that is created is small before it is destroyed. For every thread that is created, it alive for two stages. After a loop of two stages is done, the thread is destroyed and recreated again. Therefore, the diagram for this test shows no pattern.

5.6 Sorting Database with Shell Sort, Bitonic Sort, Rank Sort and Merge Sort

In this section, we apply real-world open data in the database and use selected sorting algorithms to sort those data in the fields with priority starting from “LicenseType”, “Color”, “Breed” and “DogName”. As showed in section 2, the real-world data in the database have many fields that content same values. This will affect the performance of the sorting algorithm, however this factor will not be significant enough to make a certain sorting algorithm perform better than others. The data structure in the database are VARCHAR which in C language is array of characters. In order to compare them we use strcmp() function defined in the string.h header file in C programming language. This function will compare two input strings from their first character to the end and if it finds characters with different values in the input strings, it will decide their order by comparing characters that are found with different values. Since processes of comparing two strings are more complex than compare two integers in the previous tests, the comparison will take more time and post significant effect on the performance of sorting algorithms. As for swapping, in our implementation, we only swap the pointer that point to the row in the dataset so swapping operation is relatively cheap.







The x axis is the number of threads that the parallel program uses and the y axis is the execute time in seconds. The number of rows is 262144.

To discuss the result of this test, we start by looking at the following data which recorded the amount of swapping and comparison operations that have been conducted by these four algorithms when sorting data with data size 1MB (1024*256 elements) and with 16 duplicated elements for each unique value.

	Shell Sort	Bitonic Sort	Merge Sort	Rank Sort
Swapping	8014788	10517003	N/A	262144
Comparison	11904223	40370175	4387071	68721838126

In the first diagram, Bitonic Sort takes more than two times longer compare with the Shell Sort. However, in previous tests, the execution time between Bitonic Sort and Shell Sort does not have this significant difference. In the above table, we can see the number of comparison operations that the Bitonic Sort do is 3.3 times bigger than the Shell Sort. As discussed earlier in this section, the comparison of strings will take much more time than comparing integers. Therefore, with the longer execution time for every comparison operation, the difference in performance of the Bitonic Sort and Shell Sort becomes bigger. Additionally, we can find that the performance of Rank Sort is worst. In the above table, Rank Sort has least amount of swapping operations and conduct most amount of comparisons. This shows that comparing to the comparison operation the swapping operation is cheaper. As a result, even with least amount of swapping operations conducted, Rank Sort still has worst performance due to the amount of comparison it done.

6 Conclusion

- Merge Sort has the Best Performance

  From the strong scaling performance tests, we can find that Merge Sort has the best performance in the six selected sorting algorithms. However, one down side of Merge Sort in this implementation is it needs to allocate an extra storage space for the merge process. Therefore, with multiple threads running at same time, the peak memory usage is the double of the dataset size.

- Parallel Version of Sorting Algorithm's Performance Depends on Its Sequential Performance

  As describe in section 4.1, the sequential performance of a sorting algorithm is the most significant factor of the performance of its parallel version. Merge Sort that has $O(n(logn))$ complexity is the best among those six and the sorting algorithms in the second group which generally have worse performance are all with $O(n^2)$ complexity. The reason behind this decisive factor is that the number of processors in the CPU of the testing environment is not big and the architecture of the CPU is not optimal for parallel programming. Applying these tests on many-core architecture such as GPU, might have other results. [8][9][10]

- Weak Scaling Performance Tests can be Tuned

  In the weak scaling performance tests section, we conduct our tests by increasing the dataset size and the computational resource proportionally with ratio equals to one. The results of the weak scaling performance tests are not ideal. We can tune the ratio between the dataset size and the computational resource to lower the slope. In this way, it can help to decide the amount of computational resource that we allocate for a certain size of dataset when applying to the real-world scenario.

- The Influence of Duplicated Element in Dataset

  In section 4.5, the testing data is uniform dataset with different degree of duplication. The results show that some of the sorting algorithms will be affected by different degree of duplication in test dataset while some of them don't. However, in our tests, there is no certain performance of a sorting algorithm surpass others because of this factor.

- The Number of Comparison Operations has Most Influence When Sorting Text in Database

  In our implementation, we use strcmp() function provided by C library. This makes the comparison more expensive than swapping. Therefore, the result showed in section 4.6 indicates that if an algorithm conducts more comparison than another algorithm, that algorithm will have worse performance. With this in mind, when sorting text data in databases, we should try to find a sorting algorithm that conduct less comparison and we can consider swapping operations are relatively cheap. Additionally, in our test dataset, we give fields with more duplicated elements higher priority. In this way, when comparing two strings, strcmp() function will have to iterate through all the characters in the string in order to find that the two input strings are equal and the program move on to next field. Thus, it increase execution time for program to decide the order of the two rows compared.

7 Reference

1. Darko Božidar and Tomaž Dobravec. *Comparison of parallel sorting algorithms.* Technical report November 2015
2. Clay Breshears. *The Art of Concurrency* May 2009: First Edition. ISBN: 978-0-596-52153-0
3. Henrik Swahn Faculty of Computing Blekinge Institute of Technology SE-371 79 Karlskrona Sweden. *Pthreads and OpenMP A performance and productivity study*
4. OpenMP Website: https://www.openmp.org/
5. Knuth, Donald E. (1997). "Shell's method". *The Art of Computer Programming. Volume 3: Sorting and Searching* (2nd ed.). Reading, Massachusetts: Addison-Wesley. pp. 83–95. ISBN 978-0-201-89685-5.
6. Shell, D. L. (1959). "A High-Speed Sorting Procedure" (PDF). *Communications of the ACM.* 2 (7): 30–32. doi:10.1145/368370.368387
7. K. E. BATCHER Goodyear Aerospace Corporation Akron, Ohio. *Sorting networks and their applications*
8. Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, Amit Bawaskar *GPGPU PROCESSING IN CUDA ARCHITECTURE Advanced Computing: An International Journal (ACIJ), Vol.3, No.1, January 2012*
9. Ayushi Sinha, Supervisor: Prof. Kunal Agrawal *Sorting on CUDA* August 20, 2010
10. Dmitri I. Arkhipov, Di Wu, Keqin Li, and Amelia C. Regan *Sorting with GPUs: A Survey* September 8 ,2017
11. OpenMP Architecture Review Board *OpenMP Application Programming Interface*
12. Dominik Żurek, Marcin Pietroń, Maciej Wielgosz, Kazimierz Wiatr *THE COMPARISONOF PARALLEL SORTING ALGORITHMSIMPLEMENTED ONDIFFERENT HARDWARE PLATFORMS* 2013