Parallel Odd-even Sort

Xin Huang
Dept. of CST, THU
ID: 2011011253

August 23, 2013

Abstract

Odd-even Sort like Bubble Sort, is a sorting algorithm, whose time complexity is $O(n^2)$ on a sequential machine. However, the **Odd-even Sort** can be easily parallelized on machines with multiple CPU cores, and the parallel version with time complexity $O(\frac{n^2}{m})$ on m computation nodes, is efficient on supercomputer.

This article will test the performance including the strong scalability as well as the weak scalability on the Explorer 100 machines and analyze the performance.

This is *homework 1*course*Parallel Programming*

Keyword Odd-even Sorting algorithm, parallel programming, MPI

Contents

1 Instruction

1.1 Prerequisite

This program uses *MPI* as its back end, so you **MUST** have an implementation of *MPI*, either *openmpi* or *intelmpi* is advisable.

1.2 Compilation

Invoke *make* to compile the source code. Executable is sorted in *bin/odd-even-sort*, and symbolic linked to *run/odd-even-sort*

1.3 Execution

Invoke

make run

to run with the default setting. Running parameters can be set using environment variables *NP* and *NN*, for specifying number of processes and number of numbers to be sorted respectively. Example:

NP=8 NN=100000 make run

or you can enter *run* directory and issue

mpirun -np <number of processes> ./odd-even-sort <number of numbers>

2 Foundation

2.1 Serial version

It is a comparison sort related to bubble sort, with which it shares many characteristics. It functions by comparing all (odd, even)-indexed pairs of adjacent elements in the list and, if a pair is in the wrong order (the first is larger than the second) the elements are switched. The next step repeats this for (even, odd)-indexed pairs (of adjacent elements). Then it alternates between (odd, even) and (even, odd) steps until the list is sorted.

3 Design

3.1 Assumption

1. There are n numbers need to be sorted
2. There are m processes running

3.2 Routine

Program consists of three subroutines:

- **Data Distribution** In order to make full use of the multiple cores, the program generate the numbers within each process.

  process 0 is the main process, generating and sending the random numbers to *process 1 to m*, within which using the random numbers as seeds to generate the pseudo-random-numbers for the number array.

  Each process is assigned with $\frac{n}{m}$ numbers.

  The program force that each process has the same numbers, for the convenience of the sorting routine. The superfluous number positions are filled with *MAXINT*

- **Sorting**

  I implemented two ways of sorting routings:

  - **naive Odd-even Sort**

    naive Odd-even Sort

    There are n phases in the **naive Odd-even Sorting Algorithm**. In each phase, sorting proceeded in m processes concurrently. In each process, the sorting algorithm is the serial odd-even sorting algorithm, which sorts the number starts with even and starts with odd. Every process sorts its own numbers. If the last number in a process needs comparing with other processes, the process will send its last number to the adjacent process and compare the numbers, swap if the last number is larger than the first number in the adjacent process. And then send back the smaller number to the original process. We can see that at most two numbers are needed to be exchanged between two processes per phase, and the next phase there is no need to exchange the numbers between the processes. Using the blocking send and receive is a optimal choice after sorting the pairs in the process. Then we don't need to set barriers for synchronization.

    Just as **serial naive Odd-even sort**, the numbers in array perform in the same way. The difference between the serial version and the parallel version is the array is cut and "distributed" to processes and send its last number when need to, to ensure that every phase the whole array can finish one sort phase.

- **Checking**

  Checking is finished in each process, just like sorting, send the last number in each process to the next process and comparing the last and the first in the next process. And check whether the consecutive numbers are in the correct order. In this program, I use the assertion to make sure the order of the number array is correct(from the min to the max).

4 Analysis & Result

4.1 Analysis on Time Complexity

	naive Odd-even Sort
time complexity	$O(\frac{n^2}{m})$
space complexity $O(n)$	$O(n)$
sequential version time complexity	$O(n^2)$
sequential version space complexity	$O(n)$
cost $O(n \log n)$	$O(n^2)$

Table 1: Time complexity comparison

For **naive Odd-even Sort**, apparently, it is an $O(\frac{n^2}{m})$ algorithm, for algorithm consists of n phases and each phase is $O(\frac{n}{m})$. Compared to complexity on sequential machine, which is $O(n^2)$, a multiplication factor of $\frac{1}{m}$ presented due to m process parallel computing. Thus the parallel version of naive Odd-even Sort is cost-optimal.

4.2 Result

The programs has been both tested both on Inspur TS10000 HPC Server located in Tsinghua University and my laptop. **Cluster setting:**

- CPU: Intel Xeon X5670, 2.93 GHz, 6 core
- RAM: 32(for 370 nodes)/48(for 370 nodes) GB
- Connection: InfiniBand QDR network
- OS: RedHat Linux AS 5.5
- FileSystem: LUSTRE
- Compiler: *mpic++* from *mvapich* with *icpc*[1] version 11.1

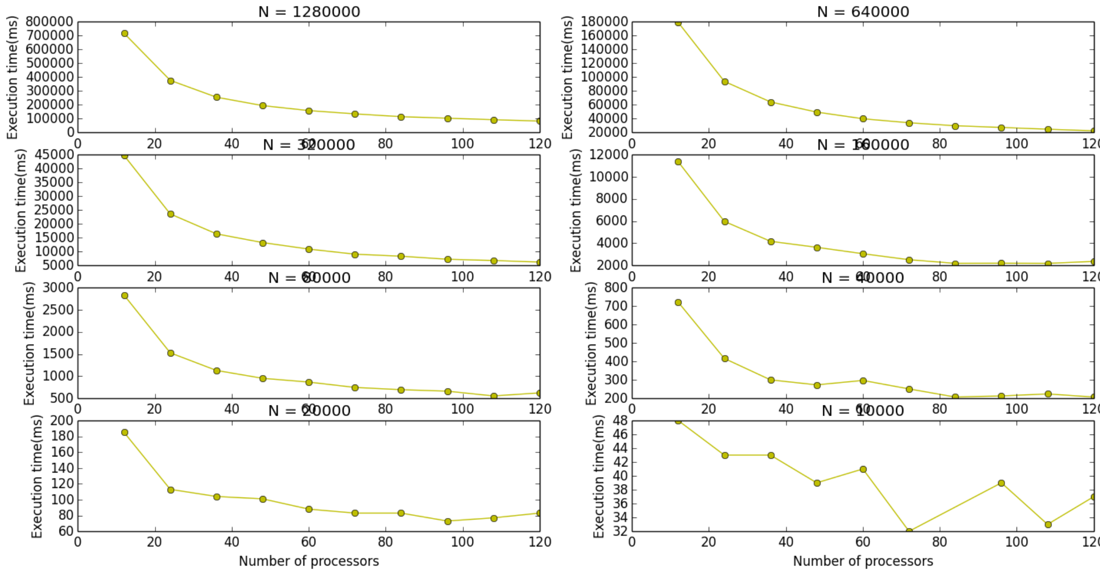
Here are the results in general:

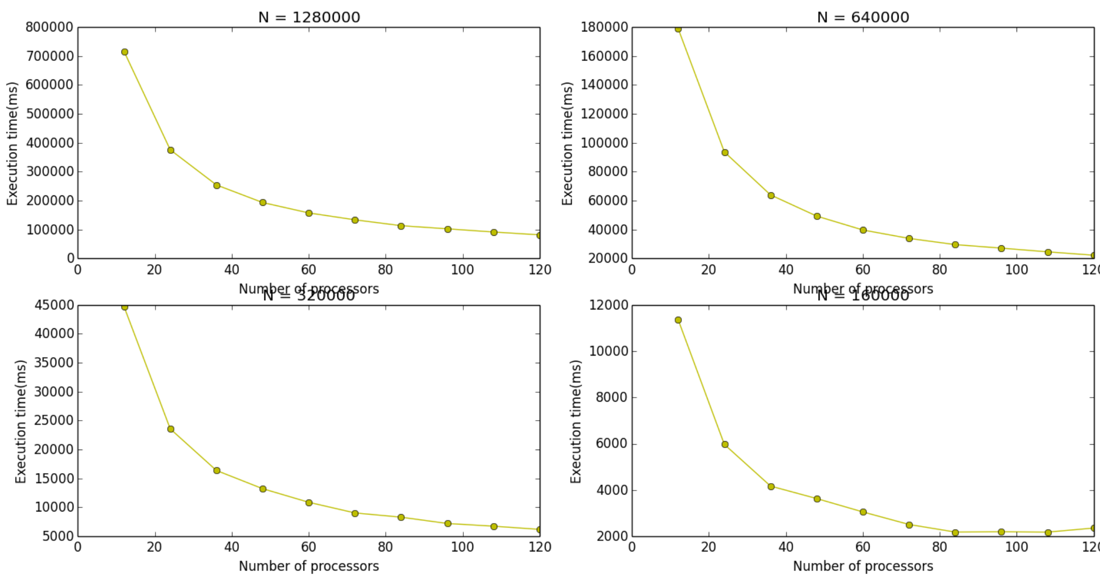
	10000	20000	40000	80000	160000	320000	640000	1280000
12	48	185	721	2829	11359	44689	178925	715801
24	43	113	416	1527	5972	23574	93713	376032
36	43	104	300	1130	4172	16384	64016	254510
48	39	101	273	952	3632	13238	49367	193756
60	41	88	297	867	3054	10883	39751	157766
72	32	83	251	746	2512	9045	33935	133943
84	33	83	207	695	2185	8310	29656	113679
96	39	73	213	662	2196	7222	27206	102834
108	33	77	224	554	2182	6755	24654	91818
120	37	83	207	623	2358	6201	22314	81926

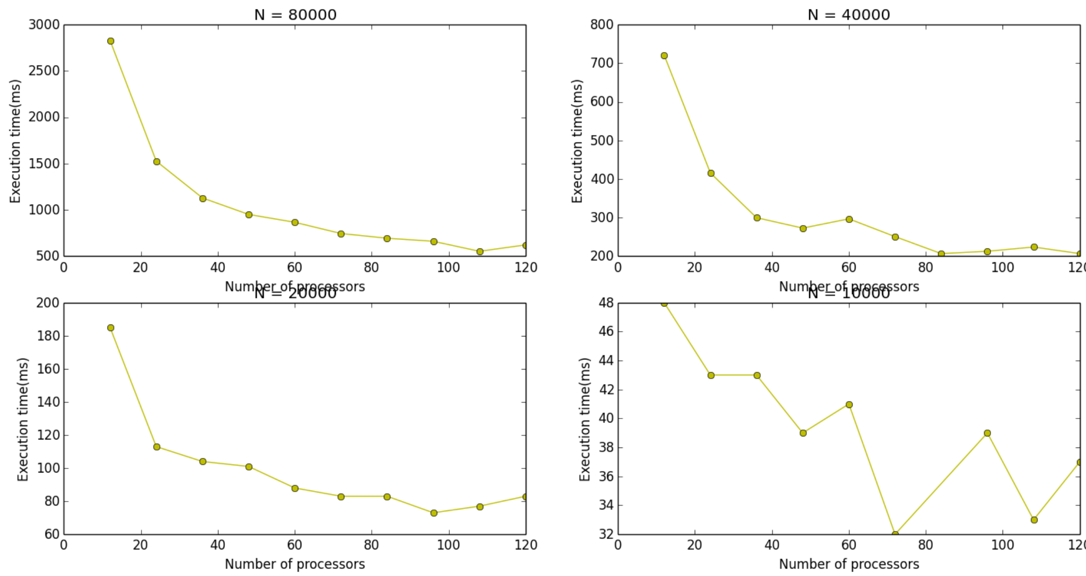
Table 2: Data-Processors Time(ms) table

[1] intel c++ compiler

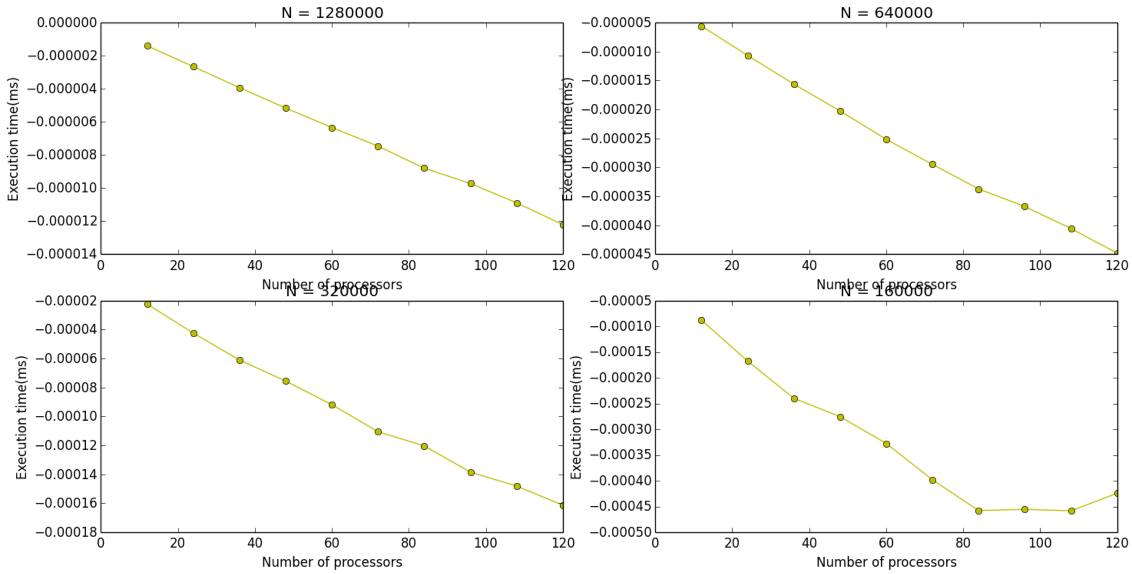
Strong Scalability: Raw data:

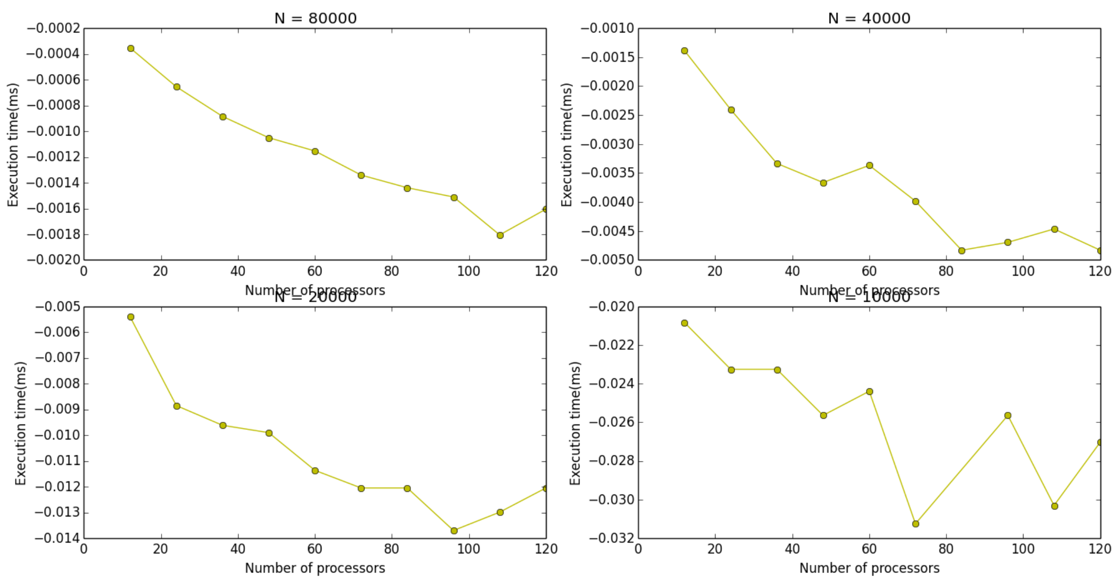




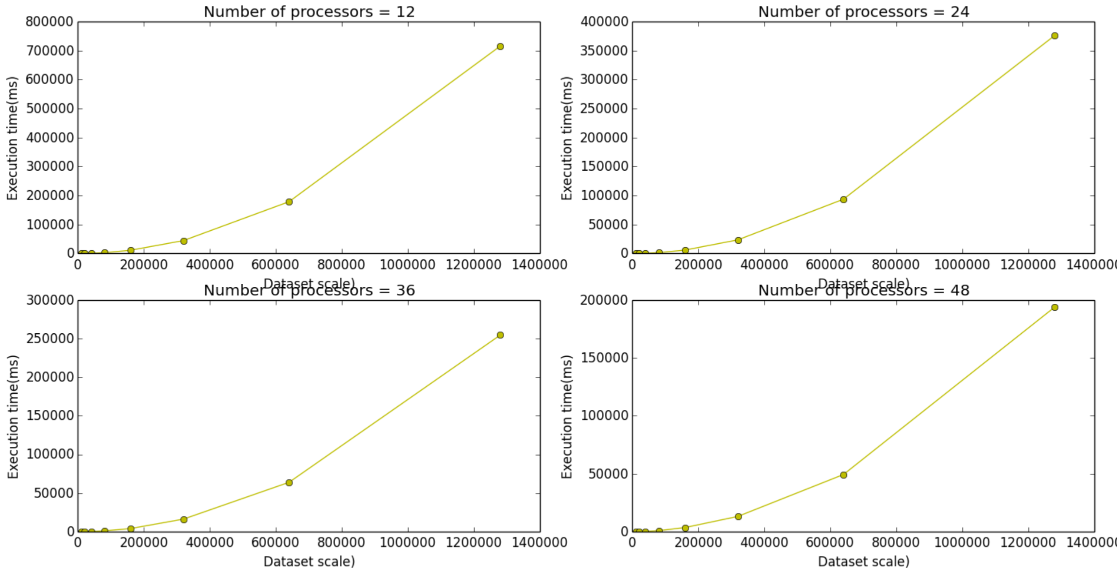

N = 80000
N = 40000
N = 20000
N = 10000
Execution time(ms)
Number of processors

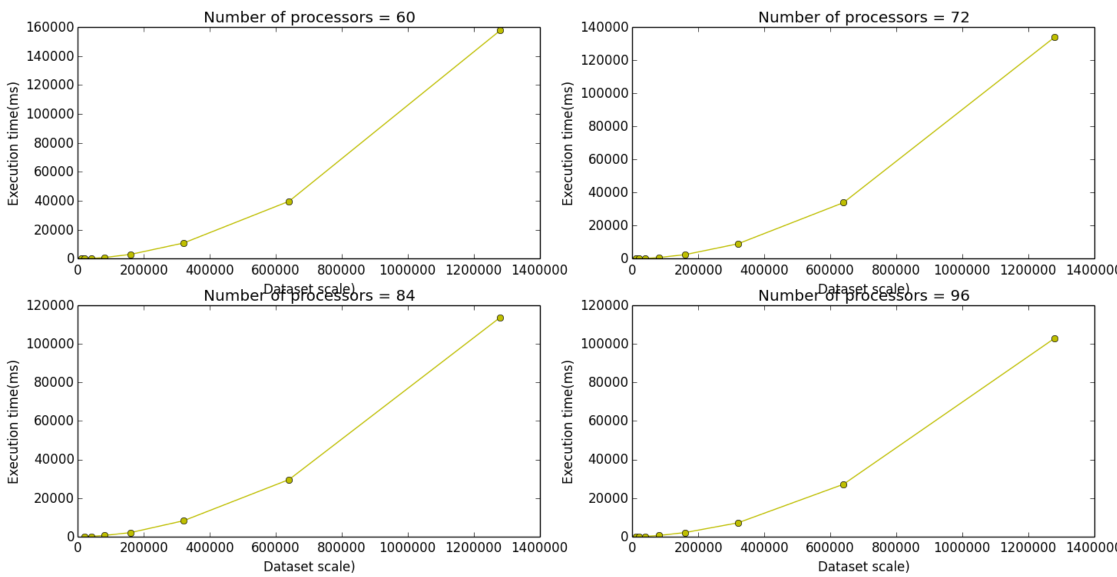
Sorting time in -1/x:

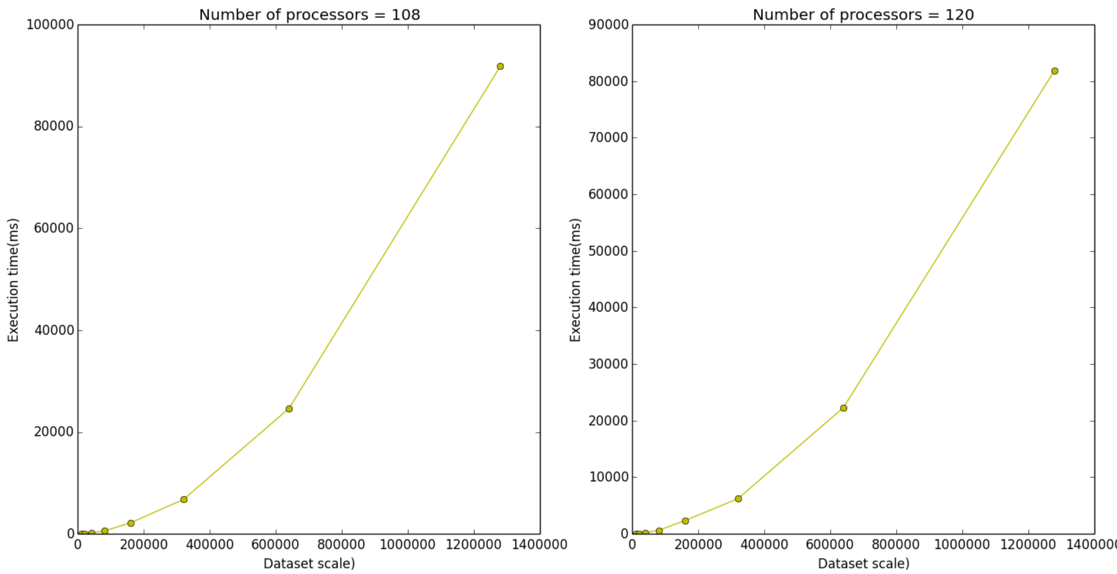




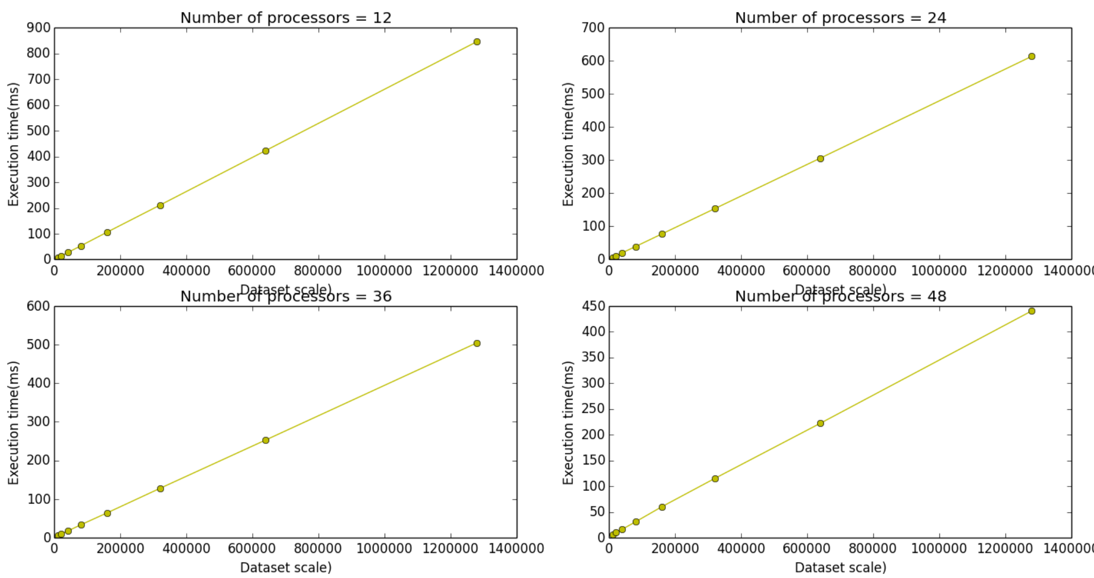
Weak Scalability: Raw data:

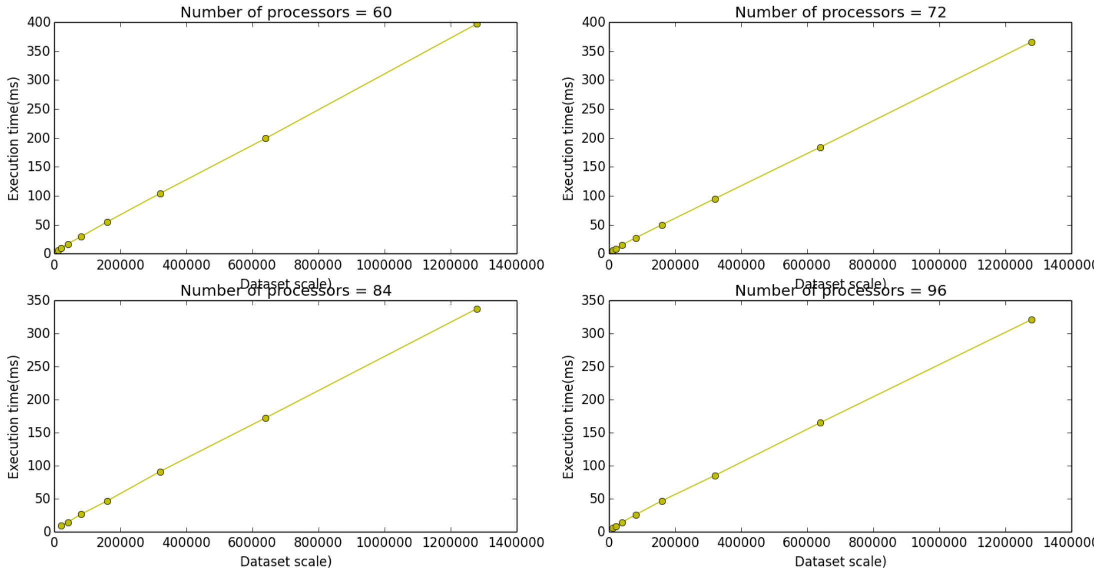


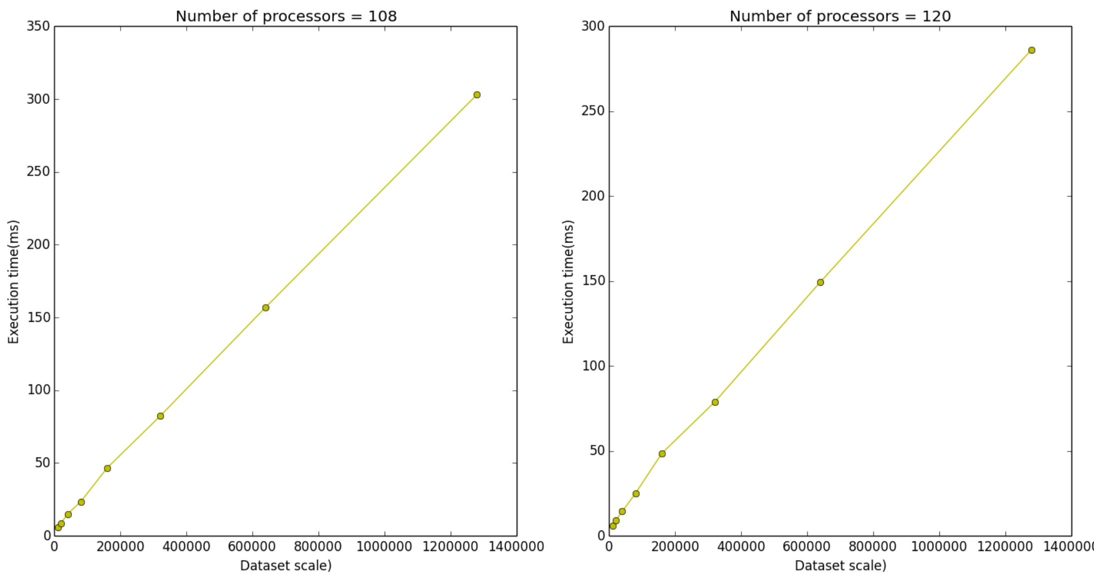




Sqaure root of the Sorting time:




Number of processors = 60
Number of processors = 72
Number of processors = 84
Number of processors = 96
Execution time(ms)
Dataset scale)
0
200000
400000
600000
800000
1000000
1200000
1400000
50
100
150
200
250
300
350
400


Number of processors = 108
Number of processors = 120
Execution time(ms)
Dataset scale)
0
200000
400000
600000
800000
1000000
1200000
1400000
50
100
150
200
250
300
350

4.3 Analysis on Result

4.3.1 Result on Clusters

The graphs illustrates the execution time with different data sets. There are some points not in their expected positions. The possible reason is that other programs running on the supercomputer takes the resources of the RAMs and CPUs.

For large data set, the time consuming is inverse proportion to the number of processors. The performance is almost linear.

For small data set, increment in time consuming as increasing number of processes may come from constant omitted in time complexity and data transfer among processes, for they dorminates the running time when data set are small. And the small data are easily to emerge the bad point, for the task may be easily stuck (I notice the Max processors and Max threads are 1 in small dataset),

4.4 Speedup Efficiency

- **Speedup** equals to serial time over the run time in m processors. According to the **Table 2**, we can see the time in ms the serial time when n = 10000 is 320ms; n = 20000 is 1280ms; n = 40000 is 5130ms. So the speedup when m = 12 almost equals to 7. And when the dataset grow lager, the speedup increase.

- **Efficiency** equals to serial to over the total time in m processors, also the **Speedup** over m. So the efficiency when m equals to 12 almost equals to 0.6. And when the dataset grow larger, the efficiency increase.

4.5 Scalability

- **Strong Scalability** means the problem size is fixed while the number of processes are increased. In strong scalability, if the speedup is equal to number of processes the program will be considered as linear scale. However, it's not very possible to achieve this goal according to the Amdahl's law. In this program, when the number of processes m increases, the numbers in each process will be less. And the percentage of communication will be larger. So the speedup will decrease. And the results illustrated support the idea. The more processes there are, the more cost of the communication, which decreases the strong scalability and the cost of time of waiting will be larger.

  In the large data set(n = 1280000, 640000, 320000 , 160000) the -1/x graphs are linear as illustration. While in the small data set(n = 80000, 40000, 20000 , 10000) the -1/x graphs are not so good(when $m > 100$) , which implicates that the strong scalability .

- **Weak Scalability** means the problem size assgin to each processing element stays constant. In weak scalability, if the runtime stays constant when the amount of data increase, the program will be considered as linear program. It's not so difficult as strong scalability to achieve. This program has a good performance in weak Scalability.

5 Experience

It's not difficult program for MPI programming. However, for it's the first time I wrote a MPI program, it took me some time to try and debug the sorting program, like tags and status settings when communication between processes.

And then I realized the cost of communication between processes is very crucial in the MPI programming. When writting a serial program I do not take this staff in my consideration. But in parallel programming, the communication may be very serious. Another odd-even Merging sorting algorithm is a faster algorithm. But the testing in my laptop shows that the bottleneck is in the transmission with lower effects.

It's very amazing to run the sorting program on the supercomputer with dozens of cores.

A Source Code

A.1 Serial Odd-even Sort

../../serial.cc

```
#include<sys/time.h>
#include<cassert>
#include"base.h"
#include"utils.h"
#include<cstdlib>
#include<iostream>

using namespace std;

int main(){
    int n = 40000;
    int *data = new int [n];
    For(i, n){
        data[i] = rand();
    }
    clock_t start = clock();
    For(p, n){
        int begin = 2 - (p&1);
        for (int i = begin; i < n; i+=2){
            if (data[i-1] > data[i]) swap(data[i-1], data[i]);
        }
    }
    cout << clock() - start << endl;
    For(i, n){
        assert(data[i-1] <= data[i]);
    }

    return 0;
}
```

A.2 Parallel Odd-even Sort

../../src/main.cc

```
#include<iostream>
#include<mpi.h>
#include"base.h"
#include"utils.h"
#include"mpiutils.h"
#include<cstdlib>
#include<cassert>

using namespace std;

int npro;
int pro_id;
int *data;
int ndata;
int npdata;

void gen(){
    if (pro_id == 0){
        for (int i = 1; i < npro; i++){
            int seed = rand();
            sendint(seed, i, 1);
        }
        for (int i = 0; i < npdata; i++){
            data[i] = rand();
        }
    }
    else{
```

```
        int seed;
        recvint(seed, 0, 1);
        srand(seed);
        for (int i = 0; i < npdata; i++){
            data[i] = rand();
        }
    }
}

void sort(){

    For(phase, ndata){
        int isodd = phase & 1;

        for (int i = isodd+1; i < npdata; i+=2){
            if (data[i-1] > data[i])
                swap(data[i-1], data[i]);
        }

        int rec;
        if (isodd == 1){
            for (int i = 0; i < 2; i++){
                if ((pro_id & 1)==i){
                    if (pro_id > 0){
                        MPI_Recv(&rec, 1, MPI_INT, pro_id-1, 0, MPI_COMM_WORLD, &status);
                        if (rec > data[0]) {swap(rec, data[0]);
                        }
                        MPI_Send(&rec, 1, MPI_INT, pro_id-1, 0, MPI_COMM_WORLD);
                    }
                }

                else{
                    if (pro_id < npro-1){
                        MPI_Send(&data[npdata-1], 1, MPI_INT, pro_id+1, 0, MPI_COMM_WORLD);
                        MPI_Recv(&data[npdata-1], 1, MPI_INT, pro_id+1, 0, MPI_COMM_WORLD, &status);
                    }
                }
            }
        }
    }
}

bool check(){
    for (int i = 1; i < npdata; i++)
        assert(data[i-1] <= data[i]);
    For(i, 2){
        if ((pro_id & 1) == i){
            if (pro_id > 0){
                int rec;
                MPI_Recv(&rec, 1, MPI_INT, pro_id-1, 2, MPI_COMM_WORLD, &status);
                assert(rec <= data[0]);
            }
        }
        else{
            if (pro_id < npro-1){
                MPI_Send(&data[npdata-1], 1, MPI_INT, pro_id+1, 2, MPI_COMM_WORLD);
            }
        }
    }
    return 1;
}




int main(int argc, char *argv[]){
    MPI_Init(&argc, &argv);

```

```
    MPI_Comm_size(MPI_COMM_WORLD, &npro);
    MPI_Comm_rank(MPI_COMM_WORLD, &pro_id);
    assert(argc == 2);
    ndata = atoi(argv[1]);
    npdata = ndata / npro;

    data = new int [npdata];

    gen();

    double start = MPI_Wtime();
    sort();
    double end = MPI_Wtime();
    bool if_chk = check();
    if (if_chk == 1 && pro_id == 0) cout << "Have sorted!" << pro_id << ":" << end-start << endl;
    MPI_Finalize();
    return 0;

}
```

../../src/base.h

```
void swap(int &a, int &b){
    int t = a; a = b; b = t;
}
```

../../src/utils.h

```
#define For(i, n) for (int i = 0; i < (n); i++)
```

B Acknowledgement

Thanks to

- Prof.Zhong and TAs for teaching this course.
- **Tsinghua University** for providing with computation resourse
- **Yeh-Ching Chung** for providing with the instruction
- **LaTeX** for typesetting
- **Python Matplotlib** for plotting data