Parallel Odd-Even Transposition Sort

By

Chen Yuan `117010038`

Submitted in fulfillment

of the requirements for

Assignment 1 of CSC4005, 2020-21 Fall Semester

THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

Tuesday 13th October, 2020

Contents

1 Introduction 1

2 Parallel Implementation Method 1

3 Results and Analysis 4

4 Conclusion 4

Appendices 6

A Code – Sequential implementation 6

B Code – MPI implementation 8

1 Introduction

This report is aiming to implement an odd-even sort algorithm by both **sequential** method (normal C++ language) and **MPI** language.

The basic odd-even sort algorithm is: for an array with n elements, it totally needs no more than n iterations. In odd iteration, the array is sorted by number in odd position comparing with adjacent the number before it (in even position), swapped the latter one is smaller than the former one. In even iteration, just opposed, the array is sorted by number in even osition comparing with the adjacent number before it. This is what I do in **sequential implementation**.

The **MPI (parallel implementation)** described in the following: when using k processes, the origional array is splited into k parts, with each having $\frac{k}{n}$ numbers. In each iteration, for number within each sub-array (in each process), first sort them by sequential odd-even methid mentioned above. Then, there are some left-over groups of numbers which should be swapped but not. So we need to check these numbers on the boundary of each group.

Fig 1 is a graph of the general idea of MPI implementation (provided by course instructor):

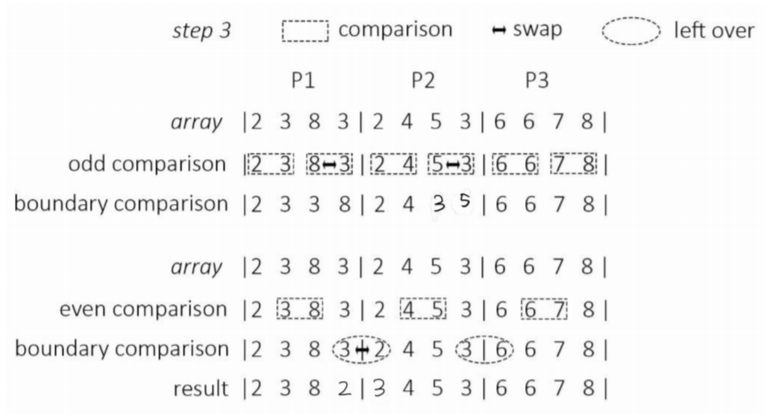

step 3
comparison
swap
left over
P1
P2
P3
array |2 3 8 3 | 2 4 5 3 | 6 6 7 8 |
odd comparison |2 3 8 3 | 2 4 5 3 | 6 6 7 8 |
boundary comparison |2 3 3 8 | 2 4 3 5 | 6 6 7 8 |
array |2 3 8 3 | 2 4 5 3 | 6 6 7 8 |
even comparison |2 3 8 3 | 2 4 5 3 | 6 6 7 8 |
boundary comparison |2 3 8 3 | 2 4 5 3 | 6 6 7 8 |
result |2 3 8 2 | 3 4 5 3 | 6 6 7 8 |

Figure 1: General idea of parallel odd-even sort

2 Parallel Implementation Method

Here are several steps:

1. (In master node) Genetate a random array of given size n. In order to avoid excessive repetition of numbers, we use $rand()\%n$ to generate random numbers within range n.

2. (In master node) Since some n is not divisible by k, in order to use **MPI_Scatter** method to distribute the array to each process, we need to add $n\%k$ max elements to the end of array, which essentially do not influence the sorting.

3. In each process, first implement the normal odd-even sort. Here we need **MPI_Barrier(MPI_COMM_WORLD)** to ensure all subarrays are done by odd-even sort. Then for each edge (border) of processes, compare them and swap value if necessary. Here are two ways: (1) for each process, send its largest (rightmost) number to next process. When the next process reveives the number, compare it with its own minimum number, then send back the ordered number to preceding process. (2) For each process, send the largest num to next process and send tne smallest num to preceding process. Then subsitite their own boundary value if necessary. (By results shown in the following, the method 2 may be quicker.)

4. In the loop of iterations, since the actual stop point may likely be less than n, we need to add a bool variable **swapped** to monitor if the array has value-swapping since the second iteration. If there is no swapping in an iteration, the loop can break. A global bool variable **global_swapped** can be calculated by "logic or" of all **swapped** in all processes. This can be achieved using **MPI_Allreduce**.

5. Finally, use **MPI_Gather** to combine all subarrays in processes to master node. Then the sort is finished.

Fig 3 shows a flow chart of the MPI program.

Oct 13, 2020.pdf Oct 13, 2020.bb

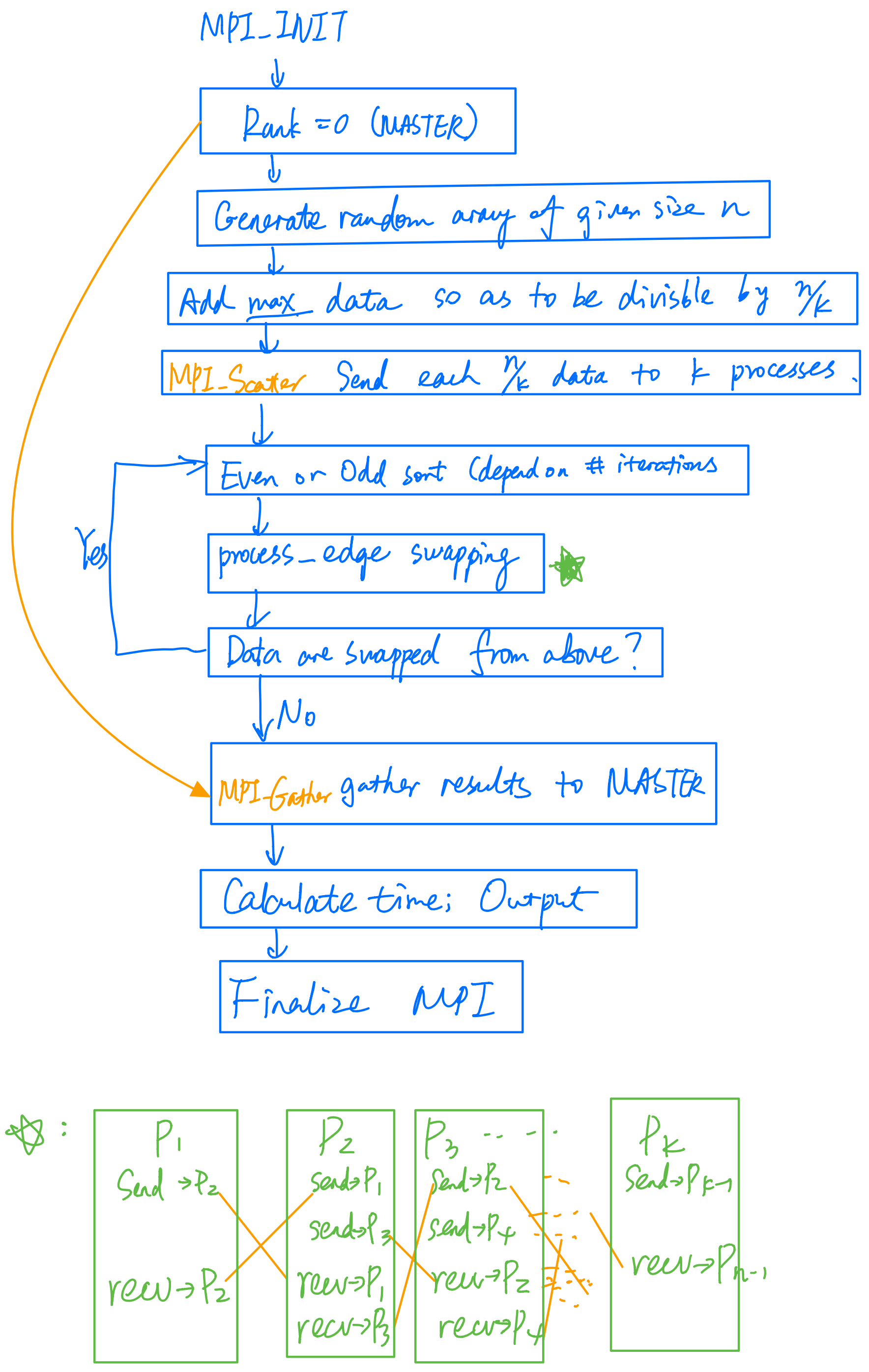

MPI_INIT
Rank =0 (MASTER)
Generate random array of given size n
Add max data so as to be divisble by n/k
MPI_Scatter Send each n/k data to k processes.
Even or Odd sort (depend on # iterations
process_edge swapping
Yes
Data are swapped from above?
No
MPI_Gather gather results to MASTER
Calculate time; Output
Finalize MPI
P1
Send → P2
recv → P2
P2
send → P1
send → P3
recv → P1
recv → P3
P3
send → P2
send → P4
recv → P2
recv → P4
Pk
Send → Pk-1
recv → Pk-1

Figure 2: flow chart of the MPI program

In the code, the parallel implementation is **sequential.cpp**; the parallel implementation is **mpi.cpp**. The implementation pbs doc is **sequential.pbs** and **parallel.pbs**, respectively. Here is a view of **parallel.pbs**:

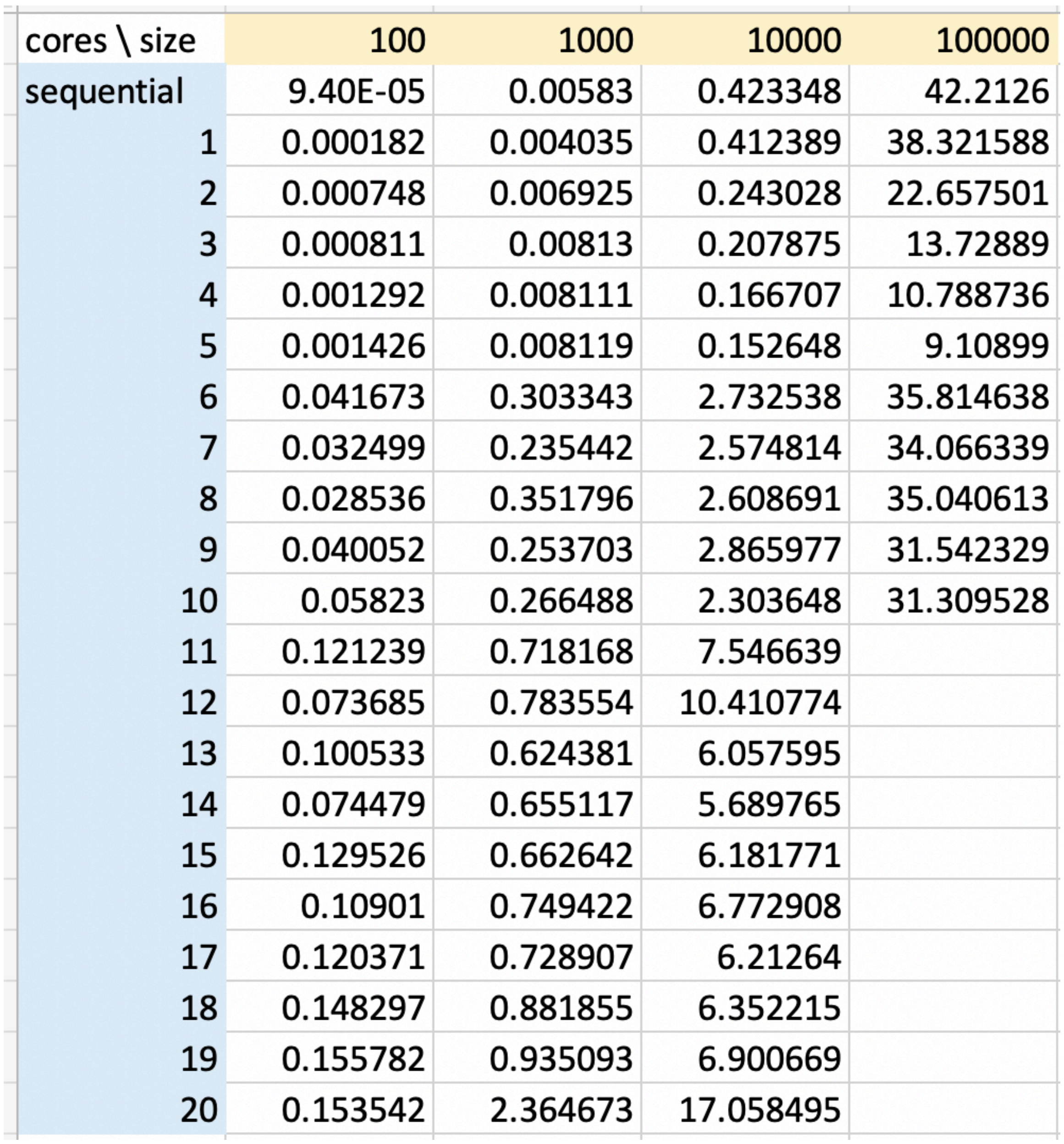
```
#!/bin/bash
#PBS -l nodes=1:ppn=5,mem=1g,walltime=00:05:00
#PBS -q batch
#PBS -m abe
#PBS -V
echo Name: Chen Yuan
echo Student ID: 117010038
echo Assignment 1, Odd-even Sort, MPI implement.
echo;
for ((n=1;n<=20;n++))
do
   timeout 60 mpirun -f /home/mpi_config -n $n /code/117010038/mpi 100
   echo;
done
```

3 Results and Analysis

The expirment conducts on $1-20$ cores; array size $n = 100, 1000, 10000, 100000$; both sequential and parallel version. (Sequential version do not have numti cores.) The following graph is the result of expriments.

4 Conclusion

The parallel can reduce the time needed for odd-even sort for large size of array, where small size of array doesn't hhave much influence. This may due to the commucation cost between processors. Indeed, the parallel version may cost much time than sequential version for small size like 100. But it has a good effect in reducting time in size greater than 10000. The cores perform good with a number less than 9, but increases time cost when using more than 15 cores.



cores \ size	100	1000	10000	100000
sequential	9.40E-05	0.00583	0.423348	42.2126
1	0.000182	0.004035	0.412389	38.321588
2	0.000748	0.006925	0.243028	22.657501
3	0.000811	0.00813	0.207875	13.72889
4	0.001292	0.008111	0.166707	10.788736
5	0.001426	0.008119	0.152648	9.10899
6	0.041673	0.303343	2.732538	35.814638
7	0.032499	0.235442	2.574814	34.066339
8	0.028536	0.351796	2.608691	35.040613
9	0.040052	0.253703	2.865977	31.542329
10	0.05823	0.266488	2.303648	31.309528
11	0.121239	0.718168	7.546639	
12	0.073685	0.783554	10.410774	
13	0.100533	0.624381	6.057595	
14	0.074479	0.655117	5.689765	
15	0.129526	0.662642	6.181771	
16	0.10901	0.749422	6.772908	
17	0.120371	0.728907	6.21264	
18	0.148297	0.881855	6.352215	
19	0.155782	0.935093	6.900669	
20	0.153542	2.364673	17.058495	

Figure 3: result