**CSC 4005 Assignment 1**

**Parallel Odd-Even Transposition Sort**

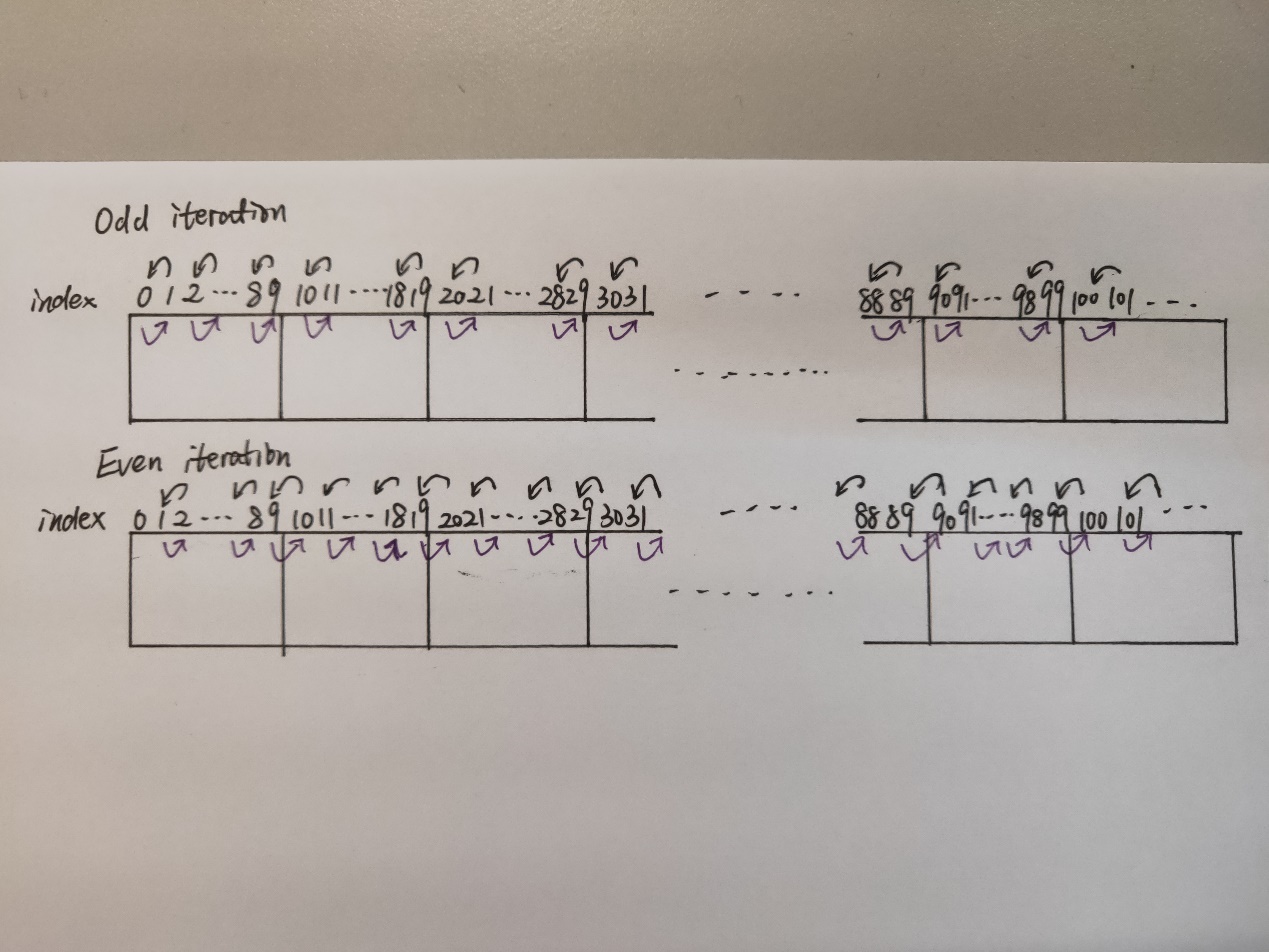
**Name: Zixuan Yao**

**Student ID: 115010267**

# Methods & Program Design

The program is required to implement Parallel Odd-Even Transposition Sort, which is to separate the original array of m elements into n processors. MPI library is used to communicate between processors.

To avoid complicated discussions of situations, I adopt a method to ensure that there’s always a even number of elements in the first n-1 processors. In this case, processors only need to communicate with their neighbors at even iteration, and do not need to communicate with each other at odd iteration, as illustrated in the figure below.



Based on this idea, the implementation of the program is intuitive. First, generate a random array of given size m. Second, scatter the array to n processors with even number of elements in first n-1 processors. Third, odd-even sort the array m times. At last, the gather all the sorted local\_array to the master processor and print the result and execution time.

After m steps, the array is guaranteed to be sorted. The detailed source code is at the end of the report.

# Instructions

Please compile the source code (attached at the end of the report) under Windows desktop C++ environment with Microsoft MPI environment. Please close the debug function when compiling, otherwise, there will be some fault preventing you from successful compile.

To run the program, please go to the direction where the compiled exe file locates and type the following command.

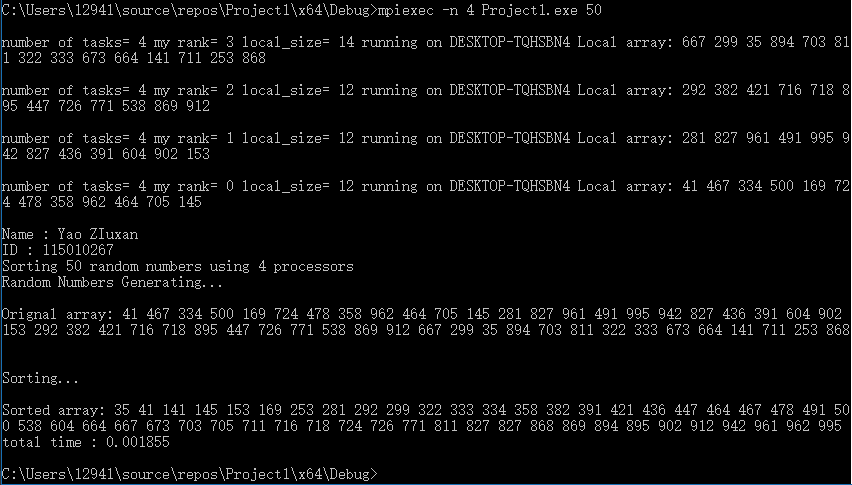


It means to run the program with four processors to sort a random array of size 40.

mpiexec -n $number of processors Project1.exe $size of array

# Results

The demo result is shown below to sort the random array of size 50 by 4 processors.



# Performance Analysis

To do the performance analysis, we need to first need to collect all the performance data and try to eliminate the variance of the data, because the time of sorting the array using MPI contains randomness. Generally, the more processors are used and the smaller the array is, the larger the relative variance is. So, I did various experience to calculate the average of the execution time and variance. The variance is shown by the black bar on the following 5 graphs.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Processors  numbers  Size  of array | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 10 | 0.000102 | 0.000728 | 0.001399 | 0.001841 | 0.002169 | 0.002384 | 0.002568 | 0.002497 |
| 100 | 0.000262 | 0.001019 | 0.001733 | 0.002022 | 0.002377 | 0.002820 | 0.003154 | 0.003563 |
| 1000 | 0.007773 | 0.005405 | 0.005543 | 0.005382 | 0.006368 | 0.006740 | 0.007023 | 0.009820 |
| 10000 | 0.716235 | 0.387081 | 0.319182 | 0.283587 | 0.254313 | 0.229669 | 0.217156 | 0.221174 |
| 100000 | 74.227552 | 39.355255 | 32.040201 | 27.333477 | 24.112805 | 21.431834 | 20.307180 | 23.231428 |

Average execution time

From the graphs, it can be easily observed that parallel computing on multi-processors won’t speed up at all when array size is small, such as 10 and 100, because the communication between processors is generally slow and takes a lot of execution time. In this case, it is faster to run the sorting on a single processor.

When the array size is increased to 1000, the execution time will decrease as the number of processors increase from 1 to 4 but will increase as the number of processors increase from 5 to 8. I think it is because my computer is physically 4-cores 8-threads by Intel hyper-thread technology.

When the array size becomes even larger, increasing to 10000 and 100000, the execution time generally decrease as the number of processors increase from 1 to 7. Adding one more processor from 7 to 8 won’t improve the performance. My explanation may be windows is still running the system and other background jobs except the sorting program, assigning task to all the 8 processors will take a little more time due to the running of windows operation system.

Furthermore, “Speed up”, “Efficiency” and “Cost” is calculated and compared.

Speed up

Efficiency

Cost

Array[10], Array [100] and Array[1000] cannot be seen from the big figure, which is shown in the following small figure.

The results of the three measures agree with the previous analysis.

The efficiency is always decreasing as the number of processors increases, because more processors consumes more sources and it is unable to superliner speedup the sorting.

The cost is always increasing as the number of processors increases, because this program is unable to superliner speedup the sorting.

# Experience

1. MPI\_Send & MPI\_Recv should be written in pair and sequentially.

If the MPI\_Send & MPI\_Recv are not written sequentially in pair, the compiler cannot correctly compile the code and the program will enter into deadlock, which this the most difficult part of writing the parallel program because you don’t know where the error is.

1. Inside MPI\_Gater & MPI\_Scatter, sendcount and recvcount should be consistent between processors.

MPI\_Gather(void\* sendbuf, int sendcount, MPI\_Datatype sendtype,

void\* recvbuf, int recvcount, MPI\_Datatype recvtype,

int root, MPI\_Comm comm)

is equilibrium to MPI\_Send by every process (including master process) to the master processor

　　MPI\_Send(sendbuf, sendcount, sendtype, root, ...),

At the same time, master process executes MPI\_Recv by n times (n is the number of process).

MPI\_Recv(recvbuf+i\*recvcount\*extent(recvtype), recvcount, recvtype, i,...),

So, sendcount and recvcount should be consistent between processors. If the number of elements you need to gather is different processors, you need to take care of that specially, or use MPI\_Gatherv function. It is the same for MPI\_Scatter.

1. When generating a random array, you need to initialize a random seed first, otherwise the random number you generate is always the same.

# Source Code

#include "mpi.h"

#include <stdio.h>

#include <memory.h>

#include <stdlib.h>

#include <string>

#include <string.h>

#include <time.h>

**using** **namespace** std**;**

int swap**(**int **\***num**,** int i**,** int j**);**

int main**(**int argc**,** char **\*** argv**[])** **{**

string argvi**(**argv**[**1**]);**

int size **=** atoi**(**argv**[**1**]);**

int **\***array\_in **=** **(**int **\*)**malloc**(sizeof(**int**)\***size**);** // array in

int **\***array\_out **=** **(**int **\*)**malloc**(sizeof(**int**)\***size**);** // array out

int local\_size**,** local\_size\_max**,** rem**;**

int recv\_right**,** send\_left**,** send\_right**,** recv\_left**;**

int numtasks**,** rank**,** len**;**

double start**,** finish**;**

double totaltime**;**

char hostname**[**MPI\_MAX\_PROCESSOR\_NAME**];**

MPI\_Init**(&**argc**,** **&**argv**);** // initialize MPI

MPI\_Comm\_size**(**MPI\_COMM\_WORLD**,** **&**numtasks**);** // get number of tasks

MPI\_Comm\_rank**(**MPI\_COMM\_WORLD**,** **&**rank**);** // get my rank

MPI\_Get\_processor\_name**(**hostname**,** **&**len**);** // this one is obvious

//Generate random array

**for** **(**int i **=** 0**;** i **<** size**;** i**++)** **{**

array\_in**[**i**]** **=** **((**int**)**rand**())** **%** 100**;**

//printf("%d ", array\_in[i]);

**}**

start **=** MPI\_Wtime**();**

rem **=** size **%** numtasks**;**

local\_size **=** size **/** numtasks**;**

local\_size\_max **=** local\_size**;**

**if** **(**local\_size **%** 2 **==** 1**)** **{** // odd numbers in each processor

local\_size **+=** 1**;**

local\_size\_max **=** local\_size**;**

**}**

**else** **{** // even numbers in each processor

**if** **(**rank **==** numtasks **-** 1**)** **{**

local\_size\_max **=** local\_size **+** rem**;**

**}**

**}**

printf**(**"\nnumber of tasks= %d my rank= %d local\_size= %d running on %s \nLocal array: "**,** numtasks**,** rank**,** local\_size\_max**,** hostname**);**

int **\***local\_array **=** **(**int **\*)**malloc**(sizeof(**int**)\***local\_size**);** // local array

**for** **(**int i **=** 0**;** i **<** local\_size\_max**;** i**++)** **{** //MPI\_Scatter(&array\_in, local\_size\_max, MPI\_INT, local\_array, local\_size\_max, MPI\_INT, 0, MPI\_COMM\_WORLD);

**if** **(**local\_size**\***rank **+** i **<** size**)** **{**

local\_array**[**i**]** **=** array\_in**[**local\_size**\***rank **+** i**];**

**}**

**else{** // padding

local\_array**[**i**]** **=** 100**;** //1000 actually should be the max of array

**}**

printf**(**"%d "**,** local\_array**[**i**]);**

**}**

**if** **(**rank **==** 0**)** **{**

printf**(**"\n\nName : Yao Zixuan\n"**);**

printf**(**"ID : 115010267\n"**);**

printf**(**"Sorting %d random numbers using %d processors\n"**,** size**,** numtasks**);**

srand**((**int**)**time**(NULL));** // random seed

printf**(**"Random Numbers Generating... \n\nOrignal array: "**);**

**for** **(**int i **=** 0**;** i **<** size**;** i**++)** **{**

printf**(**"%d "**,** array\_in**[**i**]);**

**}**

printf**(**"\n\nSorting...\n"**);**

**}**

**for** **(**int i **=** 1**;** i **<** size**+**1**;** i**++)** **{**

//printf("\n Iteration%d ", i);

**if** **(**i **%** 2 **==** 1**)** **{** // odd iteration

**for** **(**int k **=** 0**;** k **<** local\_size\_max **/** 2**;** k**++)** **{** // local\_size\_max odd, but no action

**if** **(**local\_array**[**2 **\*** k**]** **>** local\_array**[**2 **\*** k **+** 1**]){**

swap**(**local\_array**,** 2 **\*** k**,** 2 **\*** k **+** 1**);**

**}**

**}**

**}**

**else** **{** // even iteration

**if** **(**rank **==** numtasks**-**1**)** **{**

send\_left **=** local\_array**[**0**];**

MPI\_Send**(&**send\_left**,** 1**,** MPI\_INT**,** rank **-** 1**,** 0**,** MPI\_COMM\_WORLD**);**

//printf("send\_left: %d ", send\_left);

**}**

**else** **if** **(**rank **!=** 0**){**

MPI\_Recv**(&**recv\_right**,** 1**,** MPI\_INT**,** rank **+** 1**,** 0**,** MPI\_COMM\_WORLD**,** MPI\_STATUS\_IGNORE**);**

send\_left **=** local\_array**[**0**];**

MPI\_Send**(&**send\_left**,** 1**,** MPI\_INT**,** rank **-** 1**,** 0**,** MPI\_COMM\_WORLD**);**

//printf("recv\_right: %d, send\_left: %d ", recv\_right, send\_left);

**}**

**else** **if** **(**rank **==** 0**){**

**if** **(**numtasks **!=** 1**)** **{**

MPI\_Recv**(&**recv\_right**,** 1**,** MPI\_INT**,** rank **+** 1**,** 0**,** MPI\_COMM\_WORLD**,** MPI\_STATUS\_IGNORE**);**

//printf("recv\_right: %d ", recv\_right);

**}**

**}**

**for** **(**int k **=** 0**;** k **<** local\_size\_max **/** 2**;** k**++)** **{** // local\_size\_max odd

**if** **(**local\_size\_max **%** 2 **!=** 0**)** **{** // rank == numtasks - 1

**if** **(**local\_array**[**2 **\*** k **+** 1**]** **>** local\_array**[**2 **\*** k **+** 2**])** **{**

swap**(**local\_array**,** 2 **\*** k **+** 1**,** 2 **\*** k **+** 2**);**

**}**

**}**

**else** **{**

**if** **(**k **<** **(**local\_size\_max **/** 2**)** **-** 1**)** **{**

**if** **(**local\_array**[**2 **\*** k **+** 1**]** **>** local\_array**[**2 **\*** k **+** 2**])** **{**

swap**(**local\_array**,** 2 **\*** k **+** 1**,** 2 **\*** k **+** 2**);**

**}**

**}**

**else** **{**

**if** **(**rank **!=** numtasks **-** 1**)** **{**

**if** **(**local\_array**[**2 **\*** k **+** 1**]** **>** recv\_right**)** **{**

send\_right **=** local\_array**[**2 **\*** k **+** 1**];**

local\_array**[**2 **\*** k **+** 1**]** **=** recv\_right**;**

**}**

**else** **{**

send\_right **=** recv\_right**;**

**}**

**}**

**}**

**}**

**}**

**if** **(**rank **==** 0**)** **{**

**if** **(**numtasks **!=** 1**)** **{** // only one processor

MPI\_Send**(&**send\_right**,** 1**,** MPI\_INT**,** rank **+** 1**,** 0**,** MPI\_COMM\_WORLD**);**

//printf("send\_right: %d ", send\_right);

**}**

**}**

**else** **if** **(**rank **!=** numtasks **-** 1**)** **{**

MPI\_Recv**(&**recv\_left**,** 1**,** MPI\_INT**,** rank **-** 1**,** 0**,** MPI\_COMM\_WORLD**,** MPI\_STATUS\_IGNORE**);**

local\_array**[**0**]** **=** recv\_left**;**

MPI\_Send**(&**send\_right**,** 1**,** MPI\_INT**,** rank **+** 1**,** 0**,** MPI\_COMM\_WORLD**);**

//printf("recv\_left: %d, send\_right: %d ", recv\_left, send\_right);

**}**

**else** **if** **(**rank **==** numtasks **-** 1**)** **{**

MPI\_Recv**(&**recv\_left**,** 1**,** MPI\_INT**,** rank **-** 1**,** 0**,** MPI\_COMM\_WORLD**,** MPI\_STATUS\_IGNORE**);**

local\_array**[**0**]** **=** recv\_left**;**

//printf("recv\_left: %d ", recv\_left);

**}**

**}**

**if** **(**i **==** size **-** 1**)** **{**

printf**(**"\nOutput array: "**);**

**for** **(**int i **=** 0**;** i **<** local\_size\_max**;** i**++)** **{**

printf**(**"%d "**,** local\_array**[**i**]);**

**}**

**}**

**}**

int **\***send\_rem **=** **(**int **\*)**malloc**(sizeof(**int**)\***rem**);**

int **\***recv\_rem **=** **(**int **\*)**malloc**(sizeof(**int**)\***rem**);**

MPI\_Gather**(**local\_array**,** local\_size**,** MPI\_INT**,** array\_out**,** local\_size**,** MPI\_INT**,** 0**,** MPI\_COMM\_WORLD**);**

**if** **(((**size **/** numtasks**)** **%** 2 **==** 0**)** **&&** **(**rem **>** 0**))** **{** //rem still need to send

**if** **(**rank **==** numtasks **-** 1**)** **{**

**for** **(**int i **=** 0**;** i **<** rem**;** i**++)** **{**

send\_rem**[**i**]** **=** local\_array**[**local\_size **+** i**];**

**}**

MPI\_Send**(**send\_rem**,** rem**,** MPI\_INT**,** 0**,** 0**,** MPI\_COMM\_WORLD**);**

**}**

**else** **if** **(**rank **==** 0**)** **{**

MPI\_Recv**(**recv\_rem**,** rem**,** MPI\_INT**,** numtasks **-** 1**,** 0**,** MPI\_COMM\_WORLD**,** MPI\_STATUS\_IGNORE**);**

**for** **(**int i **=** 0**;** i **<** rem**;** i**++)** **{**

array\_out**[**size **-** rem **+** i**]** **=** recv\_rem**[**i**];**

**}**

**}**

**}**

finish **=** MPI\_Wtime**();**

MPI\_Finalize**();**// done with MPI

**if** **(**rank **==** 0**)** **{**

printf**(**"\nSorted array: "**);**

**for** **(**int j **=** 0**;** j **<** size**;** j**++)** **{**

printf**(**"%d "**,** array\_out**[**j**]);**

**}**

totaltime **=** **(**double**)(**finish **-** start**);**

printf**(**"\ntotal time : %f\n"**,** totaltime**);**

**}**

**return** 0**;**

**}**

int swap**(**int **\***num**,** int i**,** int j**)** **{**

int tmp **=** num**[**i**];**

num**[**i**]** **=** num**[**j**];**

num**[**j**]** **=** tmp**;**

**return** 0**;**

**}**