

CSC 4005 (Fall 2018) Assignment 1

Student Name : Chen Yu

Student ID: 115010124

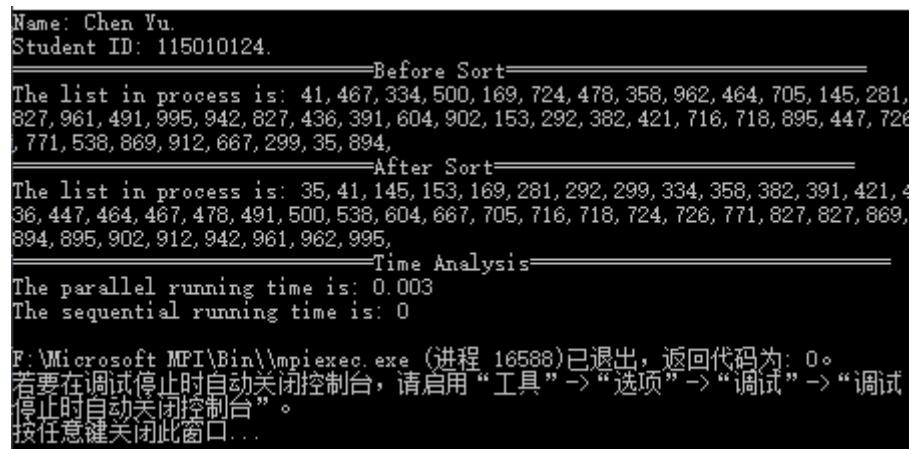
1 Instructions

Compile the C++ file `odd_even_trans_sort.cpp` under MSMPI environment. There is no other depended header or source file. Enter command:

```
mpiexec -n 4 odd_even_sort.exe
```

in cmd or powershell, `n` is the number of process. The default value of array length is determined by int variable "gobal_n" and its default value is 20000. The range of random generated integers is determined by constant int `RMAX`, and the default value is 40000.

The output structure consists of name, student ID, the unsorted array, the sorted array and time analysis. The time analysis part output the paraller running time and sequential running time for reference. The sample screen shot is showed in Figure 1.



```
Name: Chen Yu.
Student ID: 115010124.

=====Before Sort=====
The list in process is: 41,467,334,500,169,724,478,358,962,464,705,145,281,
827,961,491,995,942,827,436,391,604,902,153,292,382,421,716,718,895,447,726
,771,538,869,912,667,299,35,894,
=====After Sort=====
The list in process is: 35,41,145,153,169,281,292,299,334,358,382,391,421,4
36,447,464,467,478,491,500,538,604,667,705,716,718,724,726,771,827,827,869,
894,895,902,912,942,961,962,995,
=====Time Analysis=====
The parallel running time is: 0.003
The sequential running time is: 0

F:\Microsoft MPI\Bin\mpiexec.exe (进程 16588)已退出, 返回代码为: 0。
若要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试
停止时自动关闭控制台”。
按任意键关闭此窗口...
```

Figure 1: Sample running console.

2 Design

The goal of this assignment is learning MPI parallel library and knowing structure of parallel programming. The assignment implemented a parallel odd even transposition sort program. The

odd even sort is a variation of bubble sort with the same time complexity $O(n^2)$. However, it is easy to program in parallel, the time complexity of a parallel odd even sort is $O(n)$ theoretically. Analysis part will compare the parallel version of odd even sort with a sequential one with lists of different size. The total number of swap operations are the same for sequential and parallel version. The time difference is mainly caused by parallel speed up and communication delay.

The program can generate random number or read a list from a plaintext file by function *Gnerate_list*. This global array is saved at the master process, rank 0. *MPI_Scatter* and *MPI_Gather* function can send or receive local arrays with a certain length to all the other slave processes. For convenience, the global array must be evenly divisible by process number. In parallel iterations, the local array is sorted by local odd even sort function at the same time with communication swap among processors. The length of local arrays is even, so, swap between processor can only occur in odd phase. All the process except rank 0 send its first element to its left neighbour process and receive the last element of it. If the last element is larger than the first element of its right neighbour process, the subarray should be updated, vice versa. There is no communication in even phase.

3 Experiment & Analysis

To check the performance of parallel odd even sort, several experiments were conducted. Different number of processes and array were tested. Test platform is showed in Table. 1.

Table 1: Platform

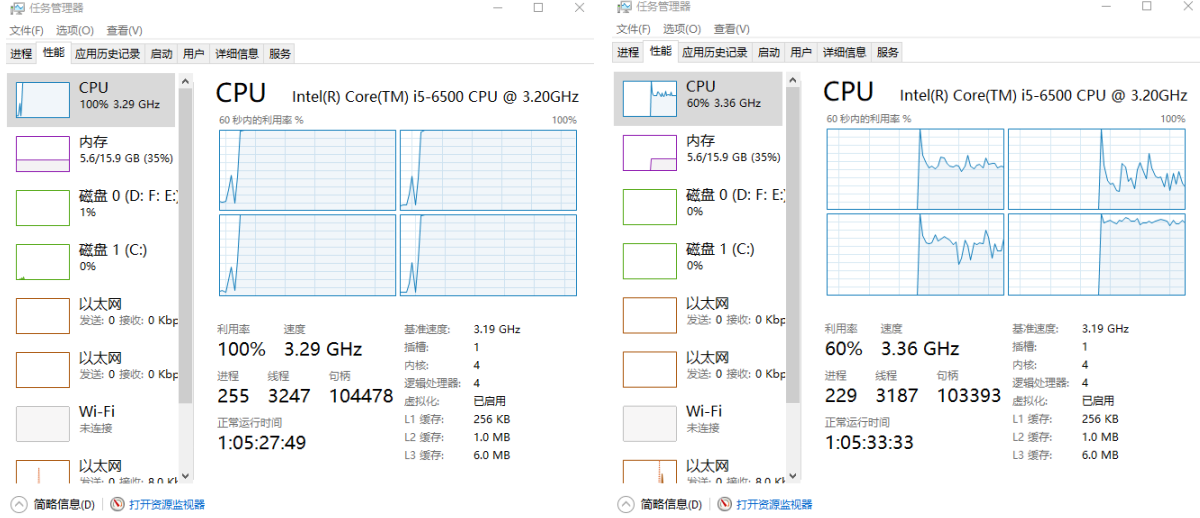
CPU	Intel(R) Core(TM) i5-6500 @3.20GHz
Memory	16.0G dual channel @2133MHz
System	Windows professional 64-bit version 1803
Compiler	Visual Studio 2017

The standard library `time.h` is included to calculate running time. The program uses clock to get the accurate time. The running time printed on screen is the sorting time without random number generating progress. The equation converting clocks to running time is:

$$T = (t_e - t_s) * f_c \quad (1)$$

Here, t_e and t_s are end and start time stamps, f_c is clock frequency.

When the program is running, CPU utilization is monitored by operating system. It shows in Figure. 2 that 4 or more processes can fully occupy the CPU. 2 processes can only use around 60% resource. One of 4 cores had a special high utilization. Text in figures are in Chinese due to default language of my personal computer.



(a) 4 processes utilization

(b) 2 processes utilization

Figure 2: Different processes utilization.

Test results are showed on Table. 2. The row in table is process number and column is length of array. Values in the table are the average time of 5-time tests. It is noticeable that when array length reach 100k, the parallel sorting algorithm finally perform better than sequential in the first time.

Table 2: Test Result

Arr/Core	1	2	4	8	16
1k	0.001	0.003	0.019	1.536	2.964
4k	0.016	0.020	0.047	4.892	10.794
40k	1.681	1.802	2.280	47.733	113.055
100k	9.935	10.609	6.872	125.056	279.207
400k	189.90	164.29	108.271	622.744	1063.10

For a small size of array, the performance is negative related with core number. There is always a huge gap between 4 processes and 8 processes, almost 100 times for all array lengths. Starting from 4 processes, the running time is increasing dramatically with core number, which means that the more cores the program uses, the more time the program consumes. Although the difference is not that large between 4 and 8 processes, within 4 processes, the performance is also worse than sequential program. When array length increases to 40k, the 4 processes program begins performing better than 2 processes.

The speed up factor is defined as:

$$S(n) = t_s/t_p \quad (2)$$

Here, t_s is execution time using one processor and t_p is execution time using a multiprocessor with n processors. The speed up factor for 4 processes sorting a 400k array is 1.75, the highest speed up factor. From Figure. 3 we can see that the speed up factor increases with the length of array at the same process number. The 4-cores test platform performs best at 4 processes in all the cases.

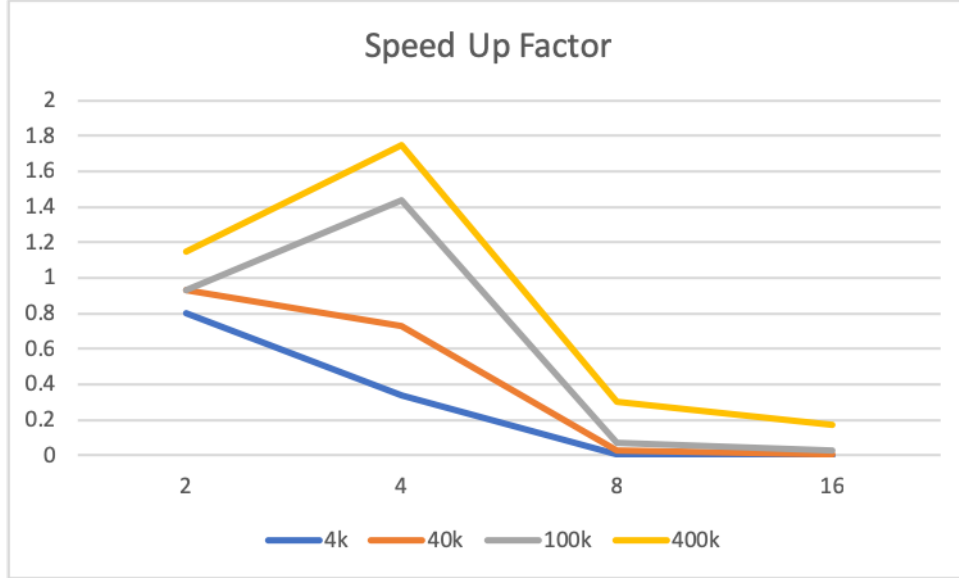


Figure 3: Speed Up Factor

The efficiency of a parallel program is defined as:

$$E = \frac{t_s}{t_p/n} \quad (3)$$

Here, t_s is execution time using one processor, t_p is execution time using a multiprocessor, n is the number of process. The efficiency is 5.783 for 4 processes sorting a 100k array.

A large array sorting is also demonstrated in experiments. A random 2 million integer between 0 to 40000 was generated to test the performance. The result shows that parallel odd even sort needs 2891.34s and sequential one needs 4377.15s. It has a 1.514 speed up factor, which is better than the 40k case. The advantage of parallel odd even transposition sort shows up when the array is large enough.

4 Experience

4.1 Distinguish process from processor

Processor is a hardware concept. It is an electronic circuit which performs operations on some external data source [1]. A process is a set of activities that interact to produce a result. It is a software multitask solution [2]. In program, if the process number is larger than processor number, it may cost more running time because of limited resource for Hyper-threading operation. The test platform of this experiment has a 4 cores with 4 threads CPU, so it performs better when n equals to 4 than 8.

4.2 Transposition V.S. Merge

Odd even transposition sort and odd even merge sort are two kinds of sorting algorithms. The merge sort combined two sorted arrays into one. The precondition for merge sort is each subarray is sorted. There is a local sorting operation before multiprocess communication in parallel version merge sort. Processes should send the whole subarray to neighbour process. After p (the number of processes) iteration, the global array is sorted.

The transposition sort communication occurs in every iteration. Not all subarrays are sorted during communication. The process only sends or receives edge elements, the first and last elements, in its own subarray. It needs n (length of global array) iteration to guarantee that the array is sorted. A while loop with a flag to indicate local swap or process communication occurring can improve average running time.

4.3 Performance

Parallel execution time includes two parts, a computation part and a communication part. [3]

$$t_p = t_{comp} + t_{comm} \quad (4)$$

For odd even transposition sort, the total arithmetic operations for sequential and parallel are the same. The parallel computation time is $1/p$ of the sequential computation time theoretically. The approximated communication time is:

$$t_{comm} = t_{startup} + n * t_{data} \quad (5)$$

Here we can see the shortcoming of the odd even transposition sort comparing to odd even merge sort. For each communication between processes, the start up time is assumed constant. The transposition sort only sends one element to neighbour process, however, merge sort sends all subarray to neighbour. Consequently, the communication time for transposition sort is much higher than merge sort.

One of the targets of this experiment is comparing parallel and sequential program with the same algorithm. We want to get a feel of "what is parallel". Although merge sort is quicker, arithmetic operation for merge sort may be highly different, The communication time is covered by the gain in the computation time. It cannot have a directly compare with the sequential one.

4.4 MPI library

- Use if loop to control certain rank process execute codes.
- Scatter and gather is easy to send arrays according to rank.
- Tags can avoid some missending cases.
- There are equations to calculate neighbour process rank.

5 Source Code

The source code is attached on file "odd-even-sort.cpp"

```
/*
 * The Chinese University of Hong Kong, Shenzhen
 * CSC4005 (2018 Fall) Assignment 1.
 * Chen Yu 115010124
 * Version 1.0
 *
 * homework1.cpp
 * Compile and run the program several times varying the number of elements
 * (global_n) and number of processes (p) using the following command:
 *      mpiexec -np <p> assignment1
 * Change RMAX = INT_MAX. Compile and run using
 *      1. global_n = 200000, 400000, 800000, 1600000, 3200000
 *      2. p from 1, 2, 4, 8.
 *      Observe how sorting time changes. Note that there will be variability
 *      in timings. It is likely that there will be substantial variation
 *      in times when this program is run with the same number of processes
 *      and same number of integers.
 *
 */

#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
```

```

#include <time.h>
#include <iomanip>

const int RMAX = 1000;

void Generate_list(int local_A[], int local_n);
void Print_list(int* Arr, int n);
void Swap(int i, int j);
int Odd_even_sort(int local_A[], const int local_n, const int my_rank, int p, MPI_Comm comm);
int sequential(int* Arr, int n);

/*-----
*/
int main(int argc, char* argv[]) {
MPI_Comm comm;
int rank; // my rank
int numProcessors; // total processors running
int gobal_n = 500000;
int local_n;
int* Arr = 0;
int n = 0;
clock_t start_time;
clock_t end_time;
clock_t start_time2;
clock_t end_time2;
double total_time;
double total_time2;

MPI_Init(&argc, &argv);
comm = MPI_COMM_WORLD;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numProcessors);

int* gobal_A = (int*)malloc(sizeof(int) * gobal_n);
local_n = gobal_n / numProcessors;

if (rank == 0)
{
Generate_list(gobal_A, gobal_n);
std::cout << "Name: Chen Yu. " << std::endl << "Student ID: 115010124." << std::endl;
std::cout << "=====Before Sort=====
<< std::endl;

```

```

Print_list(gobal_A, gobal_n);
}
int* local_A = (int*)malloc(sizeof(int) *local_n);

start_time = clock();
MPI_Scatter(gobal_A, local_n, MPI_INT, local_A, local_n, MPI_INT, 0, comm);
Odd_even_sort(local_A, local_n, rank, numProcessors, comm);
MPI_Gather(local_A, local_n, MPI_INT, gobal_A, local_n, MPI_INT, 0, comm);
end_time = clock();
total_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

if (rank == 0)
{
std::cout << "=====After Sort=====
    << std::endl;
Print_list(gobal_A, gobal_n);
std::cout << "=====Time Analysis
    =====>> std::endl;
std::cout <<"The parallel running time is: " << std::setprecision(6) << total_time <<
    std::endl;
total_time = 0;

Generate_list(Arr, n);
start_time2 = clock();
sequential(gobal_A, gobal_n);
end_time2 = clock();
total_time2 = (double)(end_time2 - start_time2) / CLOCKS_PER_SEC;
std::cout << "The sequential running time is: " << std::setprecision(6) <<
    total_time2 << std::endl;
total_time2 = 0;
}
MPI_Finalize();

return 0;
}

/*=====
* Function:    Generate_list
* Purpose:     Fill list with random ints
* Input Args: local_n
* Output Arg: local_A
*/
void Generate_list(int Gobal_A[], int gobal_n) {

```



```

int i;

srand(clock());
for (i = 0; i < gobal_n; i++) Gobal_A[i] = rand() % RMAX;
} /* Generate_list */

/*-----
* Function:    Print_list
* Purpose:     Print out master process list.
* Input Args:  Arr, n
*/
void Print_list(int* Arr, int n)
{
std::cout << "The list in process is: ";
for (int count = 0; count < n; count++) {
std::cout << Arr[count] << ", ";
}
std::cout << std::endl;
}

/*-----
* Function:    Swap
* Purpose:     Compare 2 ints, when arr[i] int is less than
*              arr[i-1], exchange these two numbers. Used by sort function.
*              if changes, return true, else return false.
*/
void Swap(int i, int j) {
int temp;

temp = i;
i = j;
j = temp;
} /* Swap */

/*-----
* Function:    Odd_even_sort
* Purpose:     Sort list, use odd-even sort to sort
*              global list.
* Input args:  local_n, my_rank, p, comm
* In/out args: local_A
*/
int Odd_even_sort(int local_A[], const int local_n, const int my_rank, int p, MPI_Comm
comm)

```

```

{
int n = local_n;
int temp = 0;
int send_temp = 0;
int recv_temp = 10001;
int rightrank = (my_rank + 1) % p;
int leftrank = (my_rank + p - 1) % p;

for (int k = 0; k < p * n; k++)
{
if (k % 2 == 0)
{
for (int j = n - 1; j > 0; j -= 2)
{
if (local_A[j] < local_A[j - 1])
{
temp = local_A[j];
local_A[j] = local_A[j - 1];
local_A[j - 1] = temp;
}
}
}
else
{
for (int j = n - 2; j > 0; j -= 2)
{
if (local_A[j] < local_A[j - 1])
{
temp = local_A[j];
local_A[j] = local_A[j - 1];
local_A[j - 1] = temp;
}
}
}
if (my_rank != 0)
{
send_temp = local_A[0];
MPI_Send(&send_temp, 1, MPI_INT, leftrank, 0, comm); //send first number to previous
process.
MPI_Recv(&recv_temp, 1, MPI_INT, leftrank, 0, comm, MPI_STATUS_IGNORE);
//if recv a number from leftrank, update local_A[0]
if (recv_temp > local_A[0]) local_A[0] = recv_temp;
}
if (my_rank != p - 1) {
int send_buff = local_A[local_n - 1];
MPI_Recv(&recv_temp, 1, MPI_INT, rightrank, 0, comm, MPI_STATUS_IGNORE);

```

```

MPI_Send(&send_buff, 1, MPI_INT, rightrank, 0, comm);
//send last number if recv number smaller than tail.
if (recv_temp < local_A[local_n - 1]) local_A[local_n - 1] = recv_temp;
}
}
}
return 0;
} /* Odd_even_sort */

```

```

/*
 * Function:    Odd_even_sort
 * Purpose:     Sort list , use sequential odd-even sort
 * Input args:  Arr, n
 */
int sequential(int* Arr, int n)
{
    int temp;
    for (int i = 0; i < n; i++)
    {
        if (n % 2 == 0)
        {
            for (int j = n - 1; j > 0; j -= 2)
            {
                if (Arr[j] < Arr[j - 1])
                {
                    temp = Arr[j];
                    Arr[j] = Arr[j - 1];
                    Arr[j - 1] = temp;
                }
            }
        }
        else
        {
            for (int j = n - 2; j > 0; j -= 2)
            {
                if (Arr[j] < Arr[j - 1])
                {
                    temp = Arr[j];
                    Arr[j] = Arr[j - 1];
                    Arr[j - 1] = temp;
                }
            }
        }
    }
}

```

```
}  
return 0;  
}
```

References

- [1] Wikipedia. Process, Wikipedia website (2018.10.26) <https://en.wikipedia.org/wiki/Process>
- [2] Wikipedia. Processor, Wikipedia website (2018.10.26) <https://en.wikipedia.org/wiki/Processor>
- [3] Barry Wilkinson, Michael Allen "Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers, Second Edition" Pearson (1999)