# CITS5507 – Project 2

Chenxin Hu (22961779)

21/10/2021

## Part1 Experiment design

### **Overview**

In this experiment, we develop the serial and parallel solutions for 3 sorting algorithms: quick sort, enumeration sort and merge sort using mpi.

Before sorting, another program used to write and read array from file is implemented using both serial and parallel methods.

The solutions for each algorithm are tested on virtual machine, using VirutalBox and Vagrant, the result of time consumption for each solution are recorded and analyzed.

### **Experiment Environment**

The experiment is carried out on a virtual machine, environment is shown below (Generated from VirtualBox):

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

### **Compile Codes**

In this experiment, we use the

mpic++ -fopenmp fileName.cpp

instruction to compile the source code.

Using the instruction to run the executable file (a.out):

mpirun -n 2 a.out or mpirun -n 4 a.out

### **Pseudocodes**

**Write & Read Array Serial Solution**

Generate a random array

Use ofstream to write it to file

Use ofstream to read it from file to a new array

Comparing 2 arrays to check the correctness

**Write & Read Array Parallel Solution**

Generate a random array

Use MPI\_File\_write to write it to file

Use serial solution to read it from file to a new array

Comparing 2 arrays to check the correctness

Use MPI\_File\_read to read it from file to a new array

Comparing 2 arrays to check the correctness

**Enumeration Sort Serial Solution**

Read array from file

Create a new empty array as temp array

For each item in the array

while array not ending

comparing the item with every item in the array.

record how many items is smaller or equal to this item

Add the item to the temp array in position number corresponding to record-1.

#Cause the item Itself takes is one of these items that “smaller or equal to the item”#

For each item in the temp array

If the item is empty #or, not fulfilled in the previous procedure

Fulfill it with the nearest filled value which locates after the item.

#Cause for one value shows many times in the array, the previous procedure only records it once in the temp array, so between the two fulfilled value, there is an empty zone. Fortunately, the empty value stands for the items have the same value but covered by the latest shown one, so the empty value should equal to the nearest fulfilled right value. #

Copy the temp array to the original array.

Delete temp array

**Enumeration Sort Parallel Solution**

Read array from file

Create a new empty array as temp array

Using MPI\_BCAST To broadcast the array to all processors

Each do the following procedure：

For each item in the array

determine the part of array it should process according to its processor id

count the rank for every item in its part

Using MPI\_SEND send back the rank message to processor 0

Processor 0 create a sorted array called data\_out using the rank message

**Quick Sort Serial Solution**

Read array from file

Create two pointer points to edge of the array

If two pointers meet or go through each other, stop the function and return

Take the value of the item that left pointer points as sample value

Create two new pointers to record the current pointer

While two new pointers do not meet each other

While new right pointer hasn’t met the value larger or equal than the sample value

Cursing left

While new left pointer hasn’t met the value smaller or equal than the sample value

Cursing right

Swap the items’ value that two new pointers point.

Create new procedure, using the original left pointer as left pointer and new right pointer as right pointer

Create new procedure, using the original right pointer as right pointer and new right pointer as right pointer

**Quick Sort Parallel Solution**

Read array from file

Using MPI\_BCAST To broadcast the array to all processors

Each do the following procedure：

If there exists processor which id=current id+2^(log2(ProcessorNum-1))

Choose a pivot value

Perform the first level iteration of serial quick sort, obtaining two unsorted arrays.

Using MPI\_SEND to send the second array to the processor which id=current id + 2^(log2(ProcessorNum-1))

Else

perform parallel quick sorting procedure on array obtained

Using MPI\_SEND to send the sorted array to the processor which send this array

**Merge Sort Serial Solution**

Read array from file

Create two pointer points to edge of the array

If two pointers meet or go through each other, stop the function and return

Break the array from the middle into two parts, apply the same procedure, using two threads to do the two parts individually

Merge two parts

**Merge Procedure**

Create a temp array

While array1 not end and array2 not end

If the item value in array1 smaller than array2

Put the value into the temp array

Right cursing array1

Else

Put the value into the temp array

Right cursing array2

Put the remaining value in array1 or array2 to temp array

Delete temp array.

**Merge Sort Parallel Solution**

Read array from file

Using MPI\_Scatter to automatically divide array to every processor

For each processor

Perform mergesort

Using MPI\_Gather to gather results from all processors to processor 0

For processor 0

Perform merge for the gathered arrays

**Merge Procedure**

Create a temp array

While array1 not end and array2 not end

If the item value in array1 smaller than array2

Put the value into the temp array

Right cursing array1

Else

Put the value into the temp array

Right cursing array2

Put the remaining value in array1 or array2 to temp array

Delete temp array.

## Part2 Experiment Result and Analysis

### Overview

In this experiment, we tested the sorting speed for each algorithm’s parallel solution and serial solution, using array size 10^3, 10^4, 10^5 and 5\*10^5. Larger array size should work, but the virtual machine execution time when the array size gets greater than 5\*10^5 become unacceptable, hence we’ll not using the array size larger than 5\*10^5.

Also, the correctness of each algorithm’s parallel solution is tested using a random array with size 10. Serial correctness is tested too.

The array used in this experiment is firstly generated and saved into file named text.txt using binary form. It is read to the storage when sorting program begins. A correctness test is done to make sure that the array read from the file is the same to the array write into the file.

The original execution screenshots, both the correctness test and speed test are in the attachment file named “screenshot.docx”

A result table is generated from the experiment:

Result Table (grouped by array size)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| AS=10^3 | 1 Core | 2 Cores | 4 Cores | SpeedUp1 | SpeedUp2 |
| ENUM | 0.005216 | 0.004002 | 0.002159 | 1.3033 | 2.2686 |
| QUICK | 0.000255 | 0.000184 | 0.000172 | 1.385 | 1.6195 |
| MERGE | 0.000156 | 0.000118 | 0.000211 | 1.1372 | 0.8257 |
|  |  |  |  |  |  |
| AS=10^4 | 1 Core | 2 Cores | 4 Cores | SpeedUp1 | SpeedUp2 |
| ENUM | 0.5281 | 0.3536 | 0.1924 | 1.4933 | 2.7345 |
| QUICK | 0.01405 | 0.01786 | 0.0023 | 0.7864 | 6.207 |
| MERGE | 0.0031 | 0.0016 | 0.0011 | 1.8788 | 2.8182 |
|  |  |  |  |  |  |
| AS=10^5 | 1 Core | 2 Cores | 4 Cores | SpeedUp1 | SpeedUp2 |
| ENUM | 53.5788 | 35.9891 | 18.6969 | 1.488 | 2.778 |
| QUICK | 1.27707 | 0.6342 | 0.6794 | 2.0134 | 1.9691 |
| MERGE | 0.0213 | 0.0125 | 0.00968 | 1.705 | 2.424 |
|  |  |  |  |  |  |
| AS=5\*10^5 | 1 Core | 2 Cores | 4 Cores | SpeedUp1 | SpeedUp2 |
| ENUM | NA | NA | NA | NA | NA |
| QUICK | 31.8047 | 31.9071 | 5.454 | 0.9967 | 5.9239 |
| MERGE | 0.1217 | 0.0618 | 0.0545 | 1.9686 | 2.2357 |

(AS stands for array size, NA means the execution time is unacceptable, it only shows in the enumeration sort when array size is larger than 10^5)

To be more obvious, we can draw an overview graph using RStudio, with x-axis stands for the array size and y-axis stands for the time consumption.

图表, 折线图

描述已自动生成

From this graph we can see that for enumeration sort, as core number grows, the sorting speed grows too. While for quick sort, when core number is small, for example, 2, the sorting speed has no significant grow then using serial solutions, the tendency of merge sort remained unknown because of scaling, we’ll look into it later.

Then we can draw another graph use speedup ratio as y-axis.

图表, 折线图

描述已自动生成

From this graph we can see that the speedup ratio for quick sort is unstable at all core numbers, the speedup ratio for merge sort and enumeration sort is stable, while merge sort’s speedup ratio on 4 cores performs less better than 2 cores comparing to enumeration sort.

We’ll investigate how parallel or serial programming influence the sorting speed detailed for each algorithm.

### Enumeration Sort Analysis

For enumeration sort, the time consumption varies by array size and core used is shown below.

图表, 折线图

描述已自动生成

From this graph we can conclude that as core grows, the sorting speed has a significant rise. The 4 core solution reaches roughly 3 times quicker when array size is 10^6, the 2 core solution reaches roughly 1.5 times quicker. For more information, we can draw a plot use speedup ratio as y-axis

图表

中度可信度描述已自动生成

From this graph we can see that the speedup ratio for mulit-cores solution reaches its maxium when array size is still small, that is, about 10000. Idealy the speedup ratio for multi-cores solution in enmuration sort should be n （the core number）. But due to the communication between cores and the start of cores will consume time, the maxium speedup ratio for 4-cores solution only reaches 2.7 and 2-cores solution only reaches 1.5, the speedup ratio for 2-cores solution even droped a little as array size grows, that may becaues the communication between cores became larger when array size is big, and it consumes more time.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | ArraySize | TimeUsed | CoreUsed | SpeedUp |
| ENUM | 1000 | 0.005216 | ONE | 1 |
| ENUM | 1000 | 0.004002 | TWO | 1.3033 |
| ENUM | 1000 | 0.002159 | FOUR | 2.2686 |
| ENUM | 10000 | 0.5281 | ONE | 1 |
| ENUM | 10000 | 0.3536 | TWO | 1.4933 |
| ENUM | 10000 | 0.1924 | FOUR | 2.7345 |
| ENUM | 100000 | 53.5788 | ONE | 1 |
| ENUM | 100000 | 35.9891 | TWO | 1.488 |
| ENUM | 100000 | 18.6969 | FOUR | 2.778 |

### Quick Sort Analysis

For quicksort, the time consumption varies by array size and core used is shown below.

图表, 折线图

描述已自动生成

The time consumption is small when core number reaches 4, but when core number is 2, the multi-core solution performs almost the same to the serial solution, to have a further view on it, we can draw a speedup ratio plot.

图表, 折线图

描述已自动生成

The tendency is more clear on speedup ratio plot, we can see the speedup ratio for multi-cores solution of quick sort is unstable, the 4-cores implementation can guarantee at least 2 times speedup while 2-cores implementation only performs better when the array size is middle.

There’re two possible reasons for this behavior, one is that the quick sort algorithm itself is an unstable sort algorithm, but it can hardly describe this behavior because both the serial and multi-core solutions use the same data set. Another possible reason is that the communication between cores consumes too much time, every iteration requires a data send and receive, so only when the array size is middle, that is less enough communication and large enough data size for serial solution to process, the multi-core solutions can perform better than serial solutions.

Observing the line shape and tendency of 4-core solutions, is similar to the 2-core solutions, we can assume that the speedup ratio will suffer a decline when the array size grows higher than 5\*10^6.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | ArraySize | TimeUsed | CoreUsed | SpeedUp |
| QUICK | 1000 | 0.000255 | ONE | 1 |
| QUICK | 1000 | 0.000184 | TWO | 1.385 |
| QUICK | 1000 | 0.000172 | FOUR | 1.6195 |
| QUICK | 10000 | 0.01405 | ONE | 1 |
| QUICK | 10000 | 0.01786 | TWO | 0.7864 |
| QUICK | 10000 | 0.0023 | FOUR | 6.207 |
| QUICK | 100000 | 1.27707 | ONE | 1 |
| QUICK | 100000 | 0.6342 | TWO | 2.0134 |
| QUICK | 100000 | 0.6794 | FOUR | 1.9691 |
| QUICK | 500000 | 31.8047 | ONE | 1 |
| QUICK | 500000 | 31.9071 | TWO | 0.9967 |
| QUICK | 500000 | 5.454 | FOUR | 5.9239 |

### Merge Sort Analysis

For merge sort, the time consumption varies by array size and core used is shown below.

图表, 折线图

描述已自动生成

From this plot we can see that the multi-cores solution acts better than the serial solutions, as core number grow, its performance also improved. Then we can draw a speedup ratio plot for a deeper sight.

图形用户界面, 图表, 折线图

描述已自动生成

From this graph we can see the speedup ratio for multi-core solutions is stable as array size grows, the ideal speedup ratio should be n (n the core number), the 2-cores solution almost reaches the maximum while 4-cores solution only performs little better than the 2-cores solution, about half its maximum. That may because the communication between cores slows the 4-cores solution down, we can predict the speedup ratio for 4-cores solution will reach its maximum as array size grows.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | ArraySize | TimeUsed | CoreUsed | SpeedUp |
| MERGE | 1000 | 0.000156 | ONE | 1 |
| MERGE | 1000 | 0.000118 | TWO | 1.1372 |
| MERGE | 1000 | 0.000211 | FOUR | 0.8257 |
| MERGE | 10000 | 0.0031 | ONE | 1 |
| MERGE | 10000 | 0.0016 | TWO | 1.8788 |
| MERGE | 10000 | 0.0011 | FOUR | 2.8182 |
| MERGE | 100000 | 0.0213 | ONE | 1 |
| MERGE | 100000 | 0.0125 | TWO | 1.705 |
| MERGE | 100000 | 0.00968 | FOUR | 2.424 |
| MERGE | 500000 | 0.1217 | ONE | 1 |
| MERGE | 500000 | 0.0618 | TWO | 1.9686 |
| MERGE | 500000 | 0.0545 | FOUR | 2.2357 |

## Part3 Conclusion

For enumeration sort, the speedup ratio is almost a constant as array size grows and core number grows, while for quick sort algorithm, its maximum speedup ratio is high, but it can only reach its maximum when the data set is of to a proper size, in other cases, its speedup ratio drops rapidly to a very low value.

The merge sort algorithm is like the enumeration sort, but with a higher speedup ratio maximum and reaches it linearly as the array size grown. It can also handle the large data sets while enumeration sort cannot.

In conclusion, when the data size is specified, we can pick one suitable core number to apply the quick sort algorithm for the best performance. When data set is small, the enumeration sort can be considered because it will immediately reach its best performance when data set is small. For general cases, the merge sort algorithm is the best option because it has a high and stable performance.