**CSC4005 assignment1 report**

**杨云腾**

**116010264**

How to run the program:

To compile the assignment1\_parallel.cpp: mpic++ assignment2\_parallel.cpp -o [output file]

To run the compiling output: mpirun -n [number of processes] ./[output file] [length of the array]

To compile the assignment1.cpp (sequential version): g++ assignment1.cpp -o [output file]

To run the compiling output: ./[output file]

Introduction

Odd-even sort is a kind of sorting algorithm which is basically a variation of bubble-sort. This algorithm is divided into two phases odd phase and even phase. The algorithm runs until the elements are sorted and in each iteration two phase occurs.

The following figure 1.1 shows the running process of the odd-even sort:

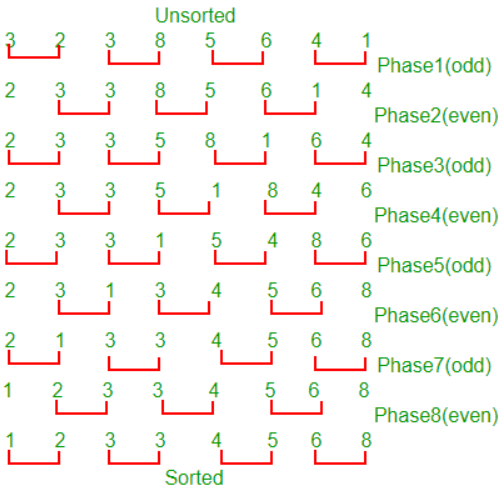


Figure 1.1

But when it is converted to a parallel version, there are some differences with sorting process (showing in figure 1.2):



Figure 1.2

To make the question running successfully on the parallel environment, I have to transfer value from one process to the other to compare values.

Design

For the parallel version of odd-even sort, I split the occasion into 4 kinds. The core implementation thought is almost the same as it is in figure 1.2. Sorting in odd sort and even sort are basically the same but with only a little bit difference.

In the first occasion in of odd sort, the index of the first element assigned to the subarray is odd and the length of the subarray is also odd, which means that there is one element at the front of the array. After finishing sorting the rest, this subarray will receive an element sent from the previous subarray. Then current array will do the comparison and send back the smaller one.

In the second occasion of add sort, the index of the first element assigned to the subarray is odd and the length of the subarray is even, which means there is one element remained at the front and one element remained at the end. If the current subarray is not the process with the largest rank, it will receive the element sent from the front and send back the smaller one and send the tail to the next process and receive the one sent back. But for the process with the largest rank, it will only receive the element sent from the smaller process but will not send the tail element to other places;

The third occasion in odd sort happens when the index of the subarray's first index is even and the subarray's length is even. It means that the array is sortable with no element left in the front or in the end. It only needs to sort the inside elements.

The last occasion in odd sort is when the index of the subarray's first element is even and the subarray's length is odd. In this occasion, if the process rank is not the largest, I need to send the tail element to the next array and receive the one sent back. Otherwise, the process only needs to sort the inside element. There is no need to send the tail element to other process.

For even sort, it is basically the same but only some occasional differences. The operation methods are the same as the methods in odd sort.

Result

This table is the result I got from testing using a machine that is familiar with the server using in class (figure 3.1):

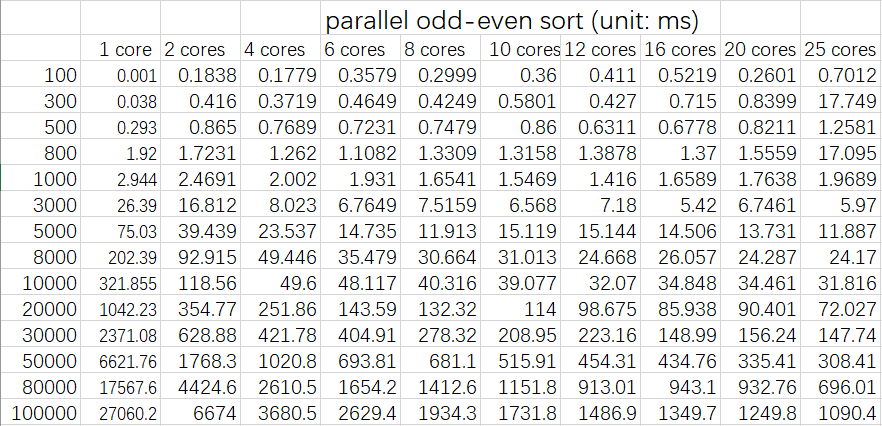


Figure 3.1

From this table (figure 3.1), it is not easy to see the difference between each result. Therefore, I need to transfer this table to line charts to analyze the results better.

Performance analysis

This is the result of using cores, from 1 core to 15 cores. It compares the result with problem size increases (figure 3.2):

Figure 3.3

From this result (figure 3.3), we can witness the changes easily. In a general trend, when we increase the problem size, it takes more time to complete the sorting algorithm. When we increase the usage of cores under the same problem size, time usage got reduced. But there are still some of the trends cannot be known from this small graph.

Now we need to analyze the data under the same problem size. The line chart in figure 3.3 cannot show the difference when the function running under the same problem size. Also, it cannot show the results when the problem size is small effectively.

Down here is the result of using 100 array length with increasing number of using cores (figure 3.3):

Figure 3.4

From the above line chart (figure 3.4), we can find that using only one core got the best performance when the problem size is small. Plus, when we use more cores, the running even got increased. When we run a problem with small problem size, the requests of swapping will not be too many. Therefore, we can finish the job with a short period using only 1 core. However, when we use more cores, MPI communication time will be counted in for the problem. Even using multicores can save time for swapping, MPI communication will still cost more time. It will not only increase the time usage but also waste computability. These cores can even be used for other computing jobs.

Here is another graph about using 50000 array length with increasing number of cores (figure 3.5):

Figure 3.5

From this graph (figure 3.5), we can easily conclude that when the problem size is far larger than core number. Increasing the running cores will reduce the run time effectively. Swapping time used for parallel computing is more costly comparing with the time used for MPI communication.

Conclusion

In this kind of purely odd-even sort, with a small problem size, sequential computing will even save time if we compute this task using only one core. The strength of parallel computing will show its power when the problem size is large. Also, when the cores are far smaller than the problem size, it will speed up the running time. However, when the cores are close to problem size, the computing speed will be lowed down.

Experience

To speed up a problem, parallel computing is useful for it, but not every problem can be speed up using parallel computing. Abusing parallel computing may not decrease the running time. It might even increase the run time. In parallel computing, we should low down the cost of cross process communication, only decrease the number of cross process communication times, faster can the problem run. In the parallel odd-even sort. If every time we sort the subarray completely, the number of communications will decrease, the program will run faster capering to current parallel computing method.