CITS5507 Project2 Report

Hanlin Zhang

22541459

Oct 16 2021

Content

1. Experimental design

1. Pseudocode of mpi quick sort algorithm

```
mpi_quick(){
    boardcast(num_value_per_process)

    create array receive, length = num_value_per_process

    scatter array_to_sort -> receive with length num_value_per_process

    quicksort(receive, 0 , num_value_per_process - 1)

    for (step = 1; step < num_of_process; step *= 2){
        if(rank % (2 * step) == 0){
        source = rank + step
            if (rource_process < num_of_process){
            recv length_of_array <- source

            create array receive, length = length_of_array

            recv array_to_merge, length = length_of_array <- source

            receive = combine receive & array_to_merge

            num_value_per_process += length_of_array
            }
        }
        else{
            target = rank - step
            Isend length_of_array -> target
            Isend receive, length = num_value_per_process -> target
            Waitall()
        }
    }
    return receive
}
```

2. Pseudocode of mpi merge sort algorithm

```
mpi_merge(){
        boardcast(num_value_per_process)

        create array receive, length = num_value_per_process

        scatter array_to_sort -> receive with length num_value_per_process

        mergesort(receive, 0 , num_value_per_process - 1)

        for (step = 1; step < num_of_process; step *= 2){
                if(rank % (2 * step) == 0){
                source = rank + step
                        if (rource_process < num_of_process){
                        recv length_of_array <- source

                        create array receive, length = length_of_array

                        recv array_to_merge, length = length_of_array <- source

                        receive = combine receive & array_to_merge

                        num_value_per_process += length_of_array
                        }
                }
                else{
                        target = rank - step
                        Isend length_of_array -> target
                        Isend receive, length = num_value_per_process -> target
                        Waitall()
                }
        }
        return receive
}
```

3. Pseudocode of mpi enum sort algorithm

```
mpi_enum(){
    boardcast(num_value_per_process)

    create array receive, length = num_value_per_process

    scatter array_to_sort -> receive with length num_value_per_process

    enumsort()

    for (step = 1; step < num_of_process; step *= 2){
        if(rank % (2 * step) == 0){
        source = rank + step
            if (rource_process < num_of_process){
            recv length_of_array <- source

            create array receive, length = length_of_array

            recv array_to_merge, length = length_of_array <- source

            receive = combine receive & array_to_merge

            num_value_per_process += length_of_array
            }
        }
        else{
            target = rank - step
            Isend length_of_array -> target
            Isend receive, length = num_value_per_process -> target
            Waitall()
        }
    }
    return receive
}
```

4. Pseudocode of parallel quick sort algorithm using omp + mpi

```
mpi_omp_quick(){
        boardcast(num_value_per_process)

        create array receive, length = num_value_per_process

        scatter array_to_sort -> receive with length num_value_per_process

        omp_quick()

        for (step = 1; step < num_of_process; step *= 2){
                if(rank % (2 * step) == 0){
                source = rank + step
                        if (rource_process < num_of_process){
                        recv length_of_array <- source

                        create array receive, length = length_of_array

                        recv array_to_merge, length = length_of_array <- source

                        receive = combine receive & array_to_merge

                        num_value_per_process += length_of_array
                        }
                }
                else{
                        target = rank - step
                        Isend length_of_array -> target
                        Isend receive, length = num_value_per_process -> target
                        Waitall()
                }
        }
        return receive
}
```

5. Pseudocode of parallel merge sort algorithm using omp + mpi

```
mpi_omp_merge(){
        boardcast(num_value_per_process)

        create array receive, length = num_value_per_process

        scatter array_to_sort -> receive with length num_value_per_process

        omp_merge()

        for (step = 1; step < num_of_process; step *= 2){
                if(rank % (2 * step) == 0){
                source = rank + step
                        if (rource_process < num_of_process){
                        recv length_of_array <- source

                        create array receive, length = length_of_array

                        recv array_to_merge, length = length_of_array <- source

                        receive = combine receive & array_to_merge

                        num_value_per_process += length_of_array
                        }
                }
                else{
                        target = rank - step
                        Isend length_of_array -> target
                        Isend receive, length = num_value_per_process -> target
                        Waitall()
                }
        }
        return receive
}
```

6. Pseudocode of parallel enum sort algorithm using omp + mpi

```
mpi_omp_enum(){
    boardcast(num_value_per_process)

    create array receive, length = num_value_per_process

    scatter array_to_sort -> receive with length num_value_per_process

    omp_enum()

    for (step = 1; step < num_of_process; step *= 2){
        if(rank % (2 * step) == 0){
        source = rank + step
            if (rource_process < num_of_process){
            recv length_of_array <- source

            create array receive, length = length_of_array

            recv array_to_merge, length = length_of_array <- source

            receive = combine receive & array_to_merge

            num_value_per_process += length_of_array
            }
        }
        else{
            target = rank - step
            Isend length_of_array -> target
            Isend receive, length = num_value_per_process -> target
            Waitall()
        }
    }
    return receive
}
```

7. Pseudocode of serial enum sort algorithm

```
enum_sort(array, array_new, array_length)
for i <– 0 to array_length
        count_smaller <– 0
        for j <– 0 to array_length
                if array[j] < array[i]
                        then count_smaller += 1
        new_array[count_smaller] <– array[i]

for i <– 0 to array_length
        if array_new[i] == 0
                then array_new[i] <– array_new[i–1]

return array_new
```

8. Pseudocode of serial merge sort algorithm

```
merge_sort(array, start, end)
        if start < end
                then half <– (start + end) / 2
                        merge_sort(array, start, half)
                        merge_sort(array, half + 1, end)
                        merge_sort_looping(array, start, end)

merge_sort_looping(array, start, end)
        half <– (start + end) / 2
        pointer <– 0
        create array temp[0, end – start + 1]
        left <– start
        right <– half + 1
```

```
    while (left <= half) and (right <= end)
        do if array[left] < array[right]
            then temp[pointer++] <– array[left++]
        else do temp[pointer++] <– array[right++]
    while(left <= half)
        do temp[pointer++] <– array[left++]
    while(right <= end)
        do temp[pointer++] <– array[right++]

    for (i <– start to end)
        array[i] <– temp[i – start]
```

9. Pseudocode of serial quick sort algorithm

```
quicksort(array, left, right)
  if(left < right)
    then pointer <– quick_sort_looping(array, left, right)
         quicksort(array, left, pointer – 1)
         quicksort(array, pointer + 1, right)

quick_sort_looping(array, left, right)
  if(left > right)
    then return 0
  temp <– array[left]
  while(left < right)
    while(left < right && array[right] >= temp)
      then right––
    array[left] <– array[right]
    while(left < right && array[left] <= temp)
      then left++
    array[right] <– array[left]
  array[left] <– temp
  return left
```

10. Pseudocode of parallel enum sort algorithm using omp

```
omp_enum(array, array_new, value)
    int count_smaller
    #omp parallel block
            #omp for avoid data race
            for i <– 0 to value
                count_smaller <– 0;
                for j <– 0 to value
                    if array[j] < array[i]
                        then do count_smaller + 1
                array_new[count_smaller] <– array[i]
            #omp for end
            #omp for
            for i <– 0 to value
                if array_new[i] = 0
                    array_new[i] <– array_new[i – 1]
            #omp for end
    #omp parallel block end
```

11. Pseudocode of parallel merge sort algorithm using omp

```
omp_merge(array, start, end)
    if start < end
          then do:
        if (end – start) > 100000
              then do:
            middle = (start + end) / 2;
            #omp parallel block
                #single thread nowait
                    task firstprivate(array, start, middle)
                    omp_merge(array, start, middle);     //array1 (left)
                    task firstprivate(array, middle, end)
                    omp_merge(array, middle + 1, end);  //array2 (right)
                #single thread nowait end
            taskwait
            then do merge_sort_looping(array, start, end)
             #omp parallel block end
        else
            merge_sort(array, start, end);
```

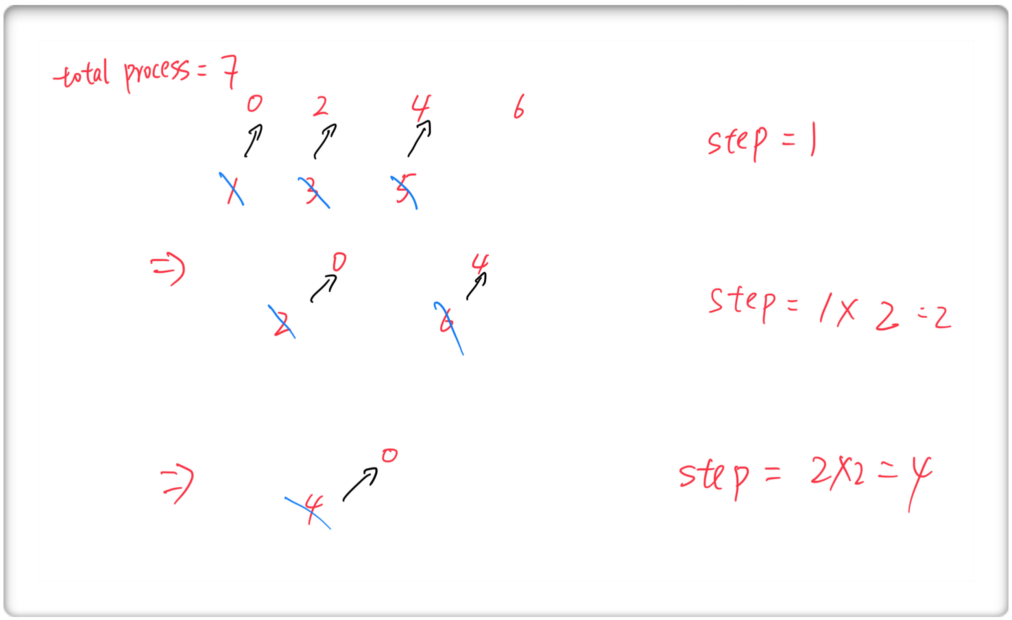
12. Pseudocode of parallel quick sort algorithm using omp

```
omp_quick(array, left, right){
    pointer = quick_sort_looping(array, left, right – 1)
    #parallel block start with 2 threads

        sections block
            section block1
            quicksort(array, 0, pointer – 1)
            section block2
            quicksort(array, pointer, right – 1)
        sections block end
    #parallel block end
```

13. Explain the experimental design

It is easy to see all of the algorithm are really similar except the algorithm we call inside (serial_quick, serial_merge, serial_enum, omp_quick, omp_merge and omp_enum). In general, the algorithm will send and receive the array from one process to another one. But if we simply send each array after sorted to process 0, we can only handle one array at the same time, so we need to design something to split the processes be two parts (one part send the array and another part receive the array). Here we can see how the for loop works to do this task:



We split the processes as two parts:

> Send processes: 1, 3, 5
> Receive processes: 0, 2, 4, 6

So, 1 –> 0, 3 –> 2, 5 –> 4, but 6 won't receive anything and waiting for receive from a sender. Due to this issue, we need to add a if statement to see if it is an odd number of processes or not. If so, won't start to receive.

After merged, the array after merged is in process 0, 2, 4 and 6.

Then we split the processes as two parts:

> Send processes: 2, 6
> Receive processes: 0, 4

2 –> 0, 6 –> 4

After merged, the array after merged is in process 0 and 4.

Finally, the array in process 4 will be send to process 0. Then , they will be merged. After doing this, the array in process 0 is an array after sorted.

For the combine function, it will combine two arrays by comparing each element in two arrays and place each element from small number to large number.

There is another important issue we need to solve in this project, that is we always suppose the array size is integer times of the process number. In other words, we always expect the program will split the task to each process equally. But what about the array size is 100k and using 7 processes? No doubt each process will get an unequally task. So, here we can use '0' to padding the array to be the multiple times of the number of process. After we sorting the array, then we can remove the 0 from array (always at the front of the array after sorted). Of course it will waste resource to sort unnecessary elements ('0' also need to be sorted), but even for number of process = 8, the maximum number of '0' we need to padding into the array is only 7, won't have a significant influence.

14. What platform this program testing

Mac M1 (gcc version:10.0), RAM 16 GB

15. How to run this program

———————————— Auto run by using bash file ——————————————

For your convenience, here add an useful .py file (auto_get_result.py) to generate a bash file to run the serial version, MPI version and MPI + OMP version together automatically. In this python file, you can modify the command you want use (sort algorithm, array size, process number and thread number to be used). Also, to make the result less error, you can increase how many times to run the command. It can use avg() to analysis in later part.

After modify the python file, run it, then you will get a bash file which named 'script.sh'. To execute this bash file, at least two ways you can do it:

```
chmod 777 script.sh
./script.sh
```

Or simpler way:

```
bash script.sh
```

Then you will see the bash file start working. Do not give a large number of array size to enum algorithm, it will spend much time to test. All the result will store in file 'sort_result.csv'.

—————————————————— Serial ——————————————————

To compile it:

```
gcc project_file.c binary.c enum.c merge.c quick.c
start_algorithm.c tools.c -o project_file
```

To run it (all will run all three algorithms):

```
./project_file array_size quick
./project_file array_size enum
./project_file array_size merge
./project_file array_size all
```

Important! Do not set a large array size when you are using enumerate sort algorithm, since the running speed will be too slow. For an array with 100000 size large might need nearly one minute to finish sorting.

To view the output, add command "print_array(array_quick, array_length)". Do not try to print the result if the array size is too large.

—————————————————— MPI ———————————————————

To compile it:

```
mpicc main.c tools.c binary.c mpi_binary.c mpi_enum.c mpi_merge.c
mpi_quick.c merge.c quick.c enum.c start_algorithm.c -o main
```

To run it (all will run all three algorithms):

```
mpirun -np process_num main array_size quick -mpi
mpirun -np process_num main array_size enum -mpi
mpirun -np process_num main array_size merge -mpi
mpirun -np process_num main array_size all -mpi
```

———————————————— MPI + OMP —————————————————

To compile it:

```
mpicc omp_mpi_main.c tools.c binary.c mpi_binary.c mpi_enum.c
mpi_merge.c mpi_quick.c merge.c quick.c enum.c start_algorithm.c
omp_merge.c omp_enum.c omp_quick.c -o omp_mpi_main
```

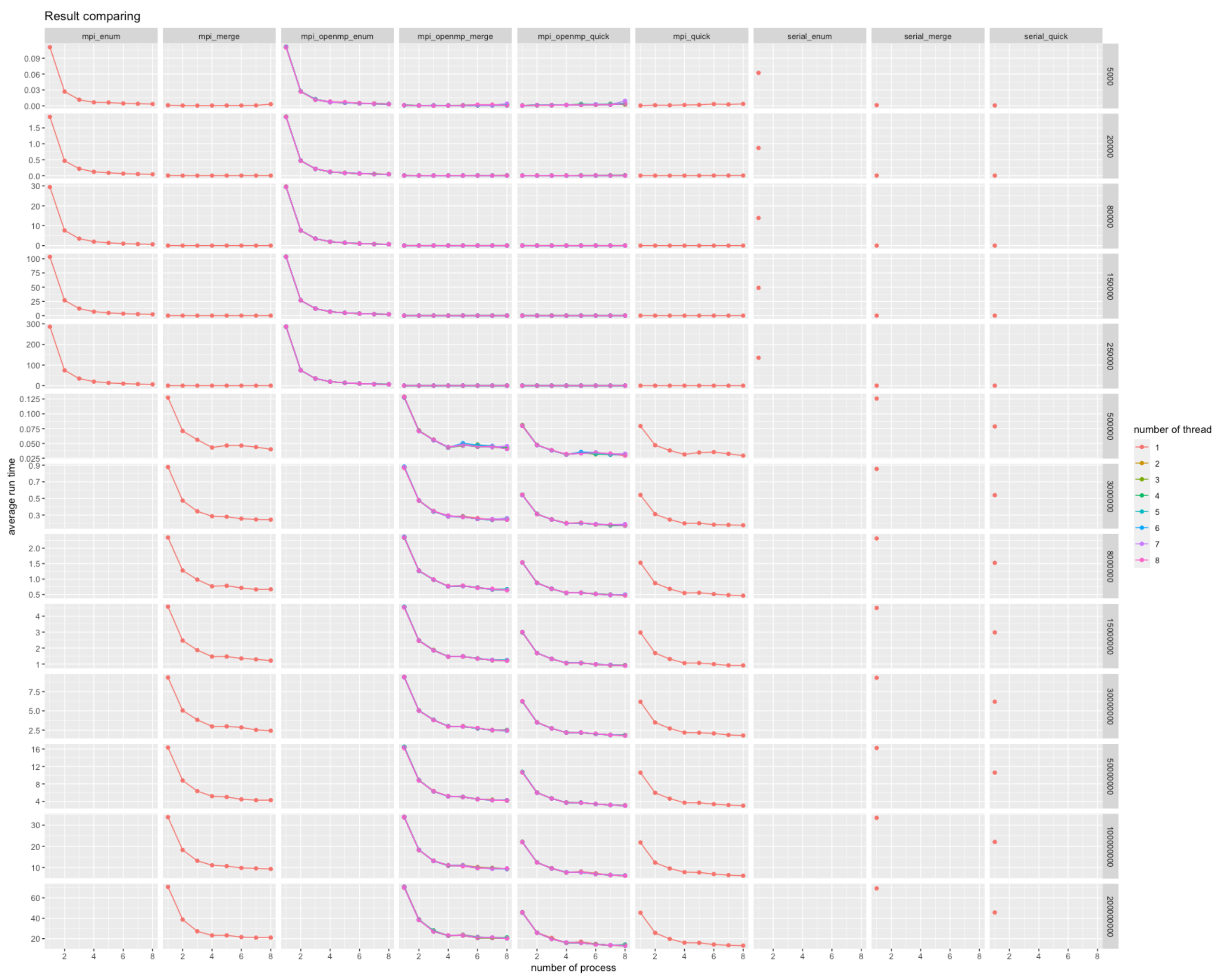
To run it (all will run all three algorithms):

```
mpirun -np process_num omp_mpi_main array_size quick thread_num
mpirun -np process_num omp_mpi_main array_size enum thread_num
mpirun -np process_num omp_mpi_main array_size merge thread_num
mpirun -np process_num omp_mpi_main array_size all thread_num
```

2. Experimental result analysis

Different with project1, here we will use the knowledge which learn from another unit to show the result more clearly and visible. The command (code) to generate the plot is showing in the file ‘output.R’. To compare the speed (time spend) between different algorithms, we can run the fun1() in R Script file, the output graph show below:

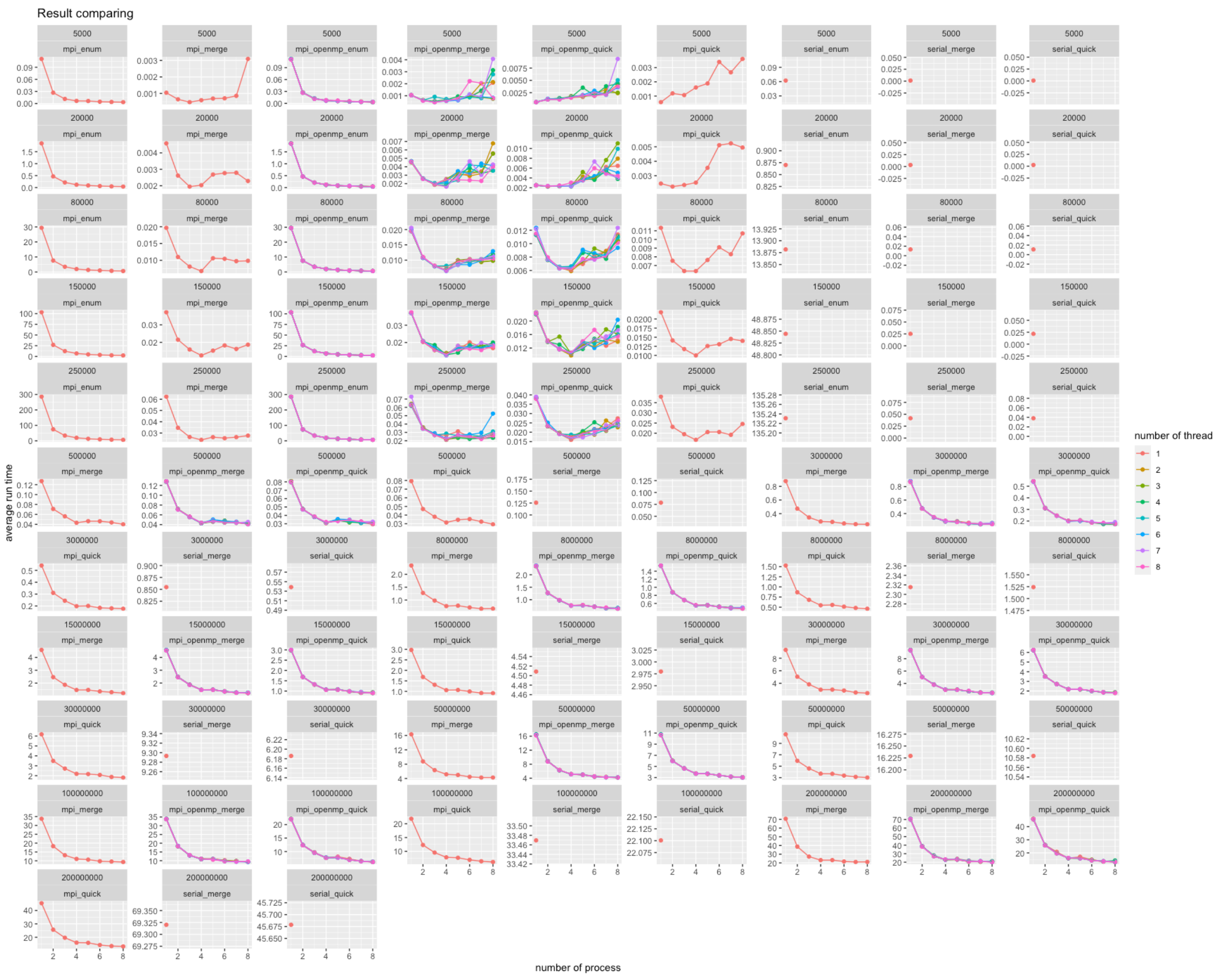
On the right hand side, you will see all of serial algorithms are all looks like “a point”, that is due to we only test serial under 1 process 1 thread (this is the real serial, no MPI and OMP inside to delay the running time). We can still compare the run time by using the points with the algorithm that using MPI or MPI+OMP.



On the same row, we use the same array size to compare between different algorithms. With increasing the number of process, we can easy to know the run time will be decreased. For both merge and quick sort (whatever MPI or MPI + OMP), the run time keep decreasing between using 1 process to 4 processes, then increase a little bit when processes number is 5. Finally, the run time continue decreased with the number of process increasing. For the enum sort (whatever MPI or MPI + OMP), it keep decreased with the increasing of the processes number.

It is really hard to compare the different sort algorithm run time by using various number of threads in this plot. So, here we can use another plot to show the trend:

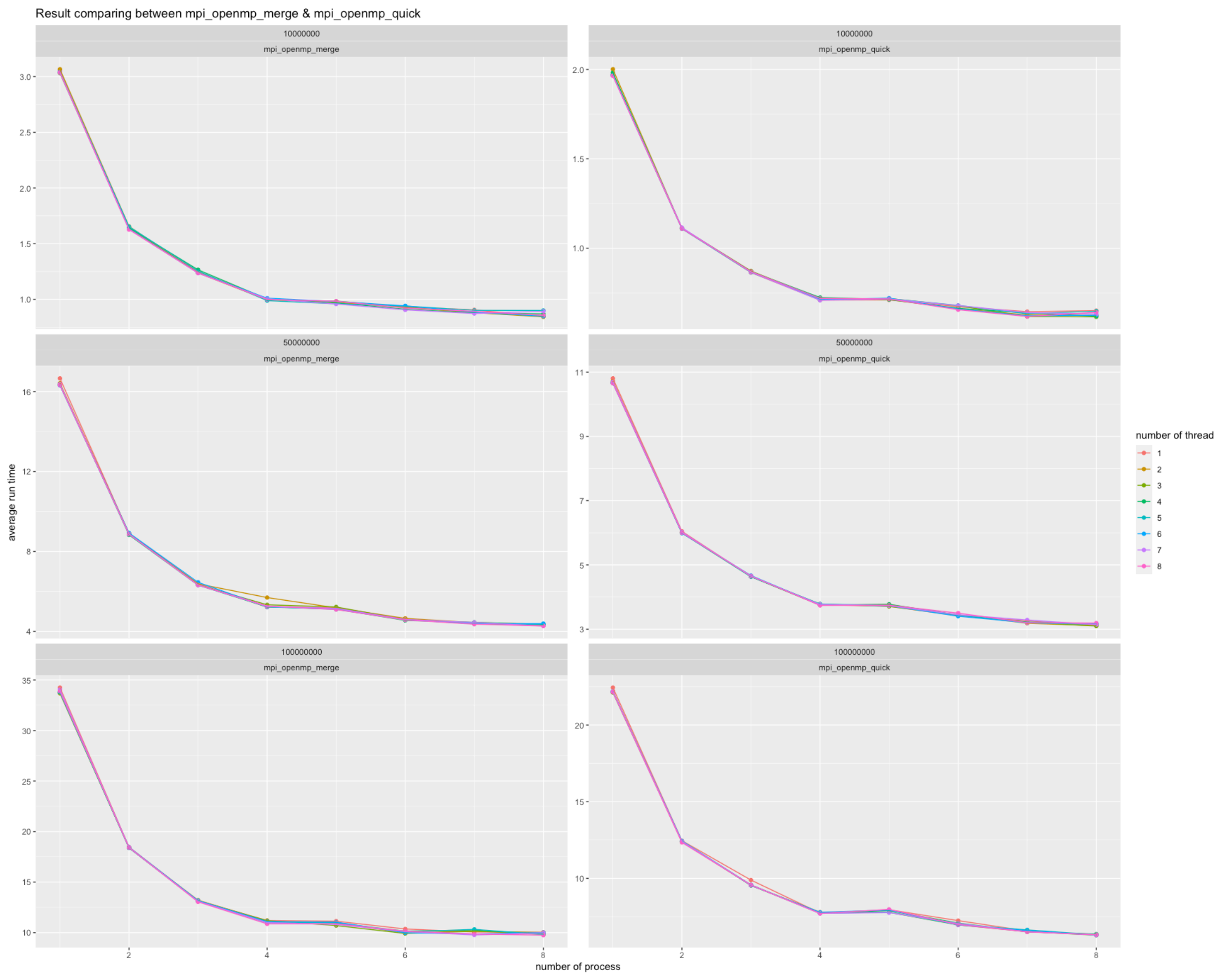
Different with the result we get from the last graph, this graph show us an important thing: when the array size is small enough (or the array size split to each process is too small), with the increasing of the number of process, the run time will also be increased, since it need cost to start a new process. So, for a small number of array size, when we use process = 4, we will get the shortest run time for both quick sort and merge sort (MPI or MPI + OMP only, not serial). Then, with a large number of array size, process = 8 is the fastest.



Compare between different number of thread, whatever for quick or merge sort (MPI or MPI + OMP only, not serial), the larger thread number solution with cost more time if the array size is too small, and cost similar time if the array size is large enough (all the lines mixed together). According to the graph, we still able to find a wired thing, the number of thread won’t increasing the speed of each algorithms (sometimes it might makes it slower by increasing the number of thread with other value fixed). Why this issue happened? Is it due to the algorithm issue?

To avoid the single experiment might contains deviation (the machine gets frozen due to some special reason etc.), so for the data we use here to plot the graphs above all test 5 times. It means for the same algorithm, array size, number of process and number of thread, there are 5 different samples to get the average running time.

But actually it is still hard to see it, is not it? So, here we can use only mpi_openmp_quick and mpi_openmp_merge to test and show the result:
After looking through the plot above, although we make the array size to be a large number, it still hard to see under how many number of thread, the algorithm will performance better (all the line plot for each thread mix together). Also, for a smaller deviation, these samples all test with 5 times and calculate the average run time.



3. Explanations of the results

By comparing between enum, quick and merge (MPI or MPI + OMP only, not serial), MPI solution will increase the running speed a lot (compare the line chart with the points plot by serial algorithms). For both quick MPI and merge MPI, with the increasing of the number of process, the run time can be 40 – 50% and 30 – 50% of serial solution. For the enum sort, the speed increasing after using MPI is more significant, the run time can be 10 – 80 % of the serial version, but it is still slower than the speed of quick sort and merge sort.

For the MPI + OMP, in the finding we show above, it is a wired thing that with the increasing of the thread and keep other values fixed, the run time might be longer, whatever for a small array size or a large one.

In general, if we gonna sort an array which size less than 80,000 – 250,000, mpi_quick will be quicker than other algorithms and the fastest way is using process = 4. For an array size larger than 250,000, mpi_quick is still the best solution. But here, we need to switch the number of process from 4 to 8.

4. How to improve the algorithm

After we get the conclusion above, there are still much stuff we can improve. Maybe we can let the program choose to use MPI / OMP or serial for a small number of array size. But actually this is the work we need to do in project 1. And also, the difference might occurs when the array size is too small (won’t get a huge influence).

Is there any way to decrease the run time of sending and receiving part? Tried to add “#pragma omp parallel num_threads(num_of_thread)” block and “#pragma omp critical” out side of the sending and receiving part, still not help. The run time of the MPI and MPI + OMP still similar, whatever how we change the thread number, it no significant change on the run time of the algorithm (test with large array size, 100 m).

5. Comment on each mpi algorithm

It is important to know why the performance of each algorithm has huge difference. On the other word, we need to know which part cause a large time spending. Testing three mpi algorithms by using different number of array size, we will see the spending on Bcast to each process or sending and receiving part not cost much time (whatever large or small size, less than 0.01 second). The only difference between these three algorithm is the serial algorithm we call inside (have a look the pseudocode of three mpi algorithms). Then we have a look the time complexity of each serial algorithm:

Algorithm	The best time complexity
Serial quick	O(n * log(n))
Serial merge	O(n * log(n))
Serial enum	O(n ^ 2)

Although after we test, the part which spend most time is compare and sorting, it is hard to reduce the time cost of all the serial algorithms, due to they matching the best time complexity.

Then we can move the target to the part that unequal split task to each process. Here, no doubt we use 0 to padding in to achieve making it looks equal split task to each process, but 0 still inside and attend sorting action. Also, sometimes this might cause some strange errors (fixing by looping the result after sort and remove the unexpected value from the result). But it is really wasting time to do this unless we can find a better way, is not it? Unfortunately, up to now, still have no good idea to fix this issue.