# CITS5507 High Performance Computing - Project 2

Wenxiao Zhang (22792191)

# Contents

# 1 Introduction

The aim of this report is to demonstrate serial and parallel solutions for each sorting algorithms including quick sort, merge sort, and enumeration sort and analyse their performance.

Four different solutions will be introduced and analysed in this report, including serial sort, parallel sort using OpenMP, parallel sort using MPI, and parallel sort using OpenMP + MPI. The experiment is carried out on a broad range of array length, and on different number of MPI processes, and on different number of threads. The ggplot will be applied to visualize the result.

# 2 Serial sort

Serial sorting algorithms are the same as we did in project 1.

## 2.1 Serial quick sort pseudocode

```
partition(array, left pointer, right pointer) {
    Initialize pivot <- left pointer;
    while (left pointer less than right pointer)

        //Move right pointer to left until the while condition is
        // not satisfied.
        while((array[right pointer] greater than or equal to pivot) and
            (right pointer not equal to left pointer))
            increment right pointer by 1;
        end while

        //Move left pointer to right until the while condition is
        // not satisfied.
        array[left pointer] <- array[right pointer];
        while((array[left pointer] less than pivot) and
            (left pointer not equal to right pointer))
            increment left pointer by 1;
        end while
        array[right pointer] <- a[left pointer];
    end while
    array[left pointer] <- pivot;
    return left pointer;
}

quickSort(array, left pointer, right pointer) {
    if (left pointer greater than or equal to right pointer)
        return;
    end if
    Initialize mid pointer <- partition(array, left pointer, right pointer);
    quickSort(array, left pointer, mid pointer);
    quickSort(array, mid pointer + 1, right pointer);
}
```

## 2.2 Serial merge sort pseudocode

```
merge(array, left index, mid index, right index) {
    Intialize temp <- a temporary memory sizes
        (right index - left index + 1)*sizeof(double);
    Intialize left pointer <- left index;
    Intialize mid pointer <- mid index;

    //pop the smaller element each time and push it into temp
    while (left pointer less than mid index and
        mid pointer less than or equal to right index )
        if (array[left pointer] less than array[mid pointer])
            Push a[left pointer] into temp;
            move left pointer to right by 1;
        else
            Push a[mid pointer] into temp;
            move mid pointer to right by 1;
        end if
    end while

    //Push remaining in the either of the two parts of the array into temp
    while(left pointer less than mid index){
        push a[left pointer] into temp;
        move left pointer to right by 1;
    }
    while(mid pointer less than right index){
        Push a[mid pointer] into temp;
        move mid pointer to right by 1;
    }

    //copy temp into the corresponding address of the array
    array[left index, right index] <- temp
    free temp;
}

mergeSort(array, left index, right index) {
    if (left index equal to right index )
        return;
    endif
    mid index = (left index + right index) / 2;
    mergeSort(array, left index, mid index);
    mergeSort(array, mid index + 1, right index);
    merge(array, left index, mid index + 1, right index);
}
```

## 2.3 Serial enumeration sort pseudocode

```
enumerationSort(array, length of the array){
    Initialize rank<-0;
    Initialize temp <- a temporary memory sizes length*sizeof(double);
    for i in range (0, length of the array -1)
        reset rank to 0;
        //count the rank of the current element
        for j in range (0, length of the array -1)
            if ((array[i] not equal to array[j]) or
                (i greater than j and a[i] equal to a[j]))
                increment rank by 1;
            end if
        end for
    end for
    copy temp into the array;
    free temp;
}
```

## 2.4 Serial sort summary

The average case run time of quick sort is O(nlogn) . This case happens when we cannot exactly get evenly balanced partitions. However, it is not a stable algorithm because the swapping of elements is done according to pivot's position, which means the worst case runtime of quick sort can be O(n^2).

Merge sort is an efficient, stable sorting algorithm with an average, best-case, and worst-case time complexity of O(nlogn). However, it takes more memory resources as it needs to allocate a new set of memory location each time when performing merge operation.

As enumeration sort has a nested for-loop, the time complextiy of which is always O(n^2), it may have the worst performance among these three algorithms.

# 3 OpenMP sort

OpenMP sorting algorithms are the same as we did in project 1.

## 3.1 OpenMP quick sort pseudocode

```
parallelQuickSort(array, left pointer, right pointer){
    if (left pointer greater than or equal to right pointer)
        return;
    endif
    Initialize mid pointer <- partition(array, left pointer, right pointer);
    //generate task
    #task directive
        //make task untied
        #untied clause
            //copy initialized mid, left pointer to every thread
            #firstprivate(mid,left)
                #if((mid pointer - left pointer) greater than 10000)
                {// executed parallelly or not is depend on the if condition
                    parallelQuickSort(array, left pointer, mid pointer);
                }
    //generate task
    #task directive
        //make task untied
        #untied clause
            //copy initialized mid, left pointer to every thread
            #firstprivate(mid, right)
                #if((right pointer-mid pointer-1) greater than 10000)
                {// executed parallelly or not is depend on the if condition
                    parallelQuickSort(array, mid pointer + 1, right pointer);
                }
}

callParallelQuickSort(array, left pointer, right pointer){
#parallel directive{
    #omp single driective{  //generate tasks
        parallelQuickSort(array, left pointer, right pointer);
    }
}
}
```

### 3.2 OpenMP merge sort pseudocode

```
parallelMergeSort(array, left index, right index) {
    if (left index equal to right index )
        return;
    endif
    //generate task
    #task directive
       //make task untied
        #untied clause
            //share the array to every thread
            #share(array)
            if ((mid index - left index) greater than 10000)
            {// executed parallelly or not is depend on the if condition
                mergeSort(array, left index, mid index);
            }
    //generate task
    #task directive
        //make task untied
        #untied clause
            //share the array to every thread
            #share(array)
                #if((right index-mid index-1) greater than 10000)
                { // executed parallelly or not is depend on the if condition
                    mergeSort(array, mid index + 1, right index);
                }
    #taskwait directive{ //execute merge after division in each recursion
            merge(array, left index, mid index + 1, right index);
    }

}

callParallelMergeSort(array, left index, right index){
#parallel directive{
    //generate tasks
    #omp single driective{
        parallelMergeSort(array, left index, right index);
    }
}

}
```

### 3.3 OpenMP enumeration sort pseudocode

```
enumerationSort(array, length of the array){
    Initialize rank <- 0;
    Initialize temp <- a temporary memory sizes length*sizeof(double);
    #parallel directive{
        //dynamically distribute iterations to each thread
        #omp for dynamic{
            set rank private to every thread{
                for i in range (0, length of the array -1)
                    reset rank to 0;
                    for j in range (0, length of the array -1)
                        if ((array[i] not equal to array[j]) or
                            (i greater than j and a[i] equal to a[j]))
                            increment rank by 1;
                        end if
                    end for
                end for
            }
        //dynamically distribute iterations to each thread
        #omp for dynamic{
            copy temp into the array;
        }
    }

    free temp;
}
```

### 3.4 OpenMP sort summary

When using OpenMP to perform parallel solution, applying task directive to quick sort and merge sort can deal with the rescursive process effectively. As for the enumeration sort, omp for and omp schedule can be applied to this sorting algorithm to dynamically distribute iterations to each thread.

parallel quick sort needs to set the mid pointer (which obtained after partition) to be firstprivate. So that each thread can have a local copy of the mid value, which can increase their performing speed and avoid race condition.

As for parallel merge sort, each thread may perform the sorting by traversing different parts of the array, so that we can make the array be shared to increase their performing speed. However, as merge sort needs to divide array up before sorting, it needs a barrier(i.e. taskwait) to achieve this.

# 4 MPI sort

The general idea of MPI sort is splitting the unsorted array evenly (except the last process when array length is not divisible by number of processes) for each processes, and each process will compute serial sorting algorithm (or OpenMP sorting algorithm when computing OMP+MPI sort) for the sub-array. After sorting completed, every other process will send their sorted sub-array to their nearby process, which will then receive and merge with its own sorted sub-array and free the space of the two sub-arrays, the completed array will finally be held in the root process.

So in this perspective, each sorting algorithm has similar MPI sorting method.

## 4.1 MPI sort pseudocode

```
mergeSubArr(array1, array2, length1, length2){
  malloc result sizes length1+length2
  for a1, a2, in array1, array2:
    if a1<a2:
      push a1 into result
    else
      push a2 into result
  return result
}

selectParallelAlgorithm(algo, subArr, subArrLen){
  if algo = "quick":
    callParallelQuickSort(subArr, 0, subArrLen-1)
  else if algo = "merge":
    callParallelMergeSort(subArr, 0, subArrLen-1)
  else if algo = "enum":
    parallelEnumerationSort(subArr, subArrLen)
}

selectAlgorithm(algo, subArr, subArrLen){
  if algo = "quick":
    quickSort(subArr, 0, subArrLen-1)
  else if algo = "merge":
    mergeSort(subArr, 0, subArrLen-1)
  else if algo = "enum":
    enumerationSort(subArr, subArrLen)
}

mpiSort(array, length){
  // calculate length of the buffer in each process (for offset)
  subBufLen = length / size

  // calculate length of the array in each process (for IO & sorting)
  if last process:
    subArrLen = length - (subBufLen * rank)
  else:
    subArrLen = subBufLen
  end if
```

```
  if rank 0 process:
    generate random array
    for i in (1, number of processes-1)

      // create array pointer to be sent to each process
      subArr = array + subBufLen*i
      MPI_send (addr = subArr, length = subArrLen, target = i)
    end for
    malloc subArr for rank 0 process
   end if

   if other processes:
     malloc subArr for receiving
     MPI_recv(addr = subArr, length = subArrLen, source = 0)
   end if

  // change this to selectParallelAlgorithm for MPI+OMP sort
  // select an algorithm to sort
  // algo: quick, merge, enum
  selectAlgorithm(algo, subArr, subArrLen)


  for step = 1; step < number of processes; step*=2:
    // send array length and array to nearby process and jump out
    if rank % (2*step) ! = 0:
      MPI_send(addr = &subArrLen, target = rank - step)
      MPI_send(addr = subArr, target = rank - step)
      break
    end if

    // receive array length and array from nearby processes
    if(rank+step < number of processes)
      MPI_Recv(addr = & recvtmpLen, source = rank + step)
      MPI_Recv(addr = recvtmp, source = rank+ step)
      //merge with array in its own process and
      //free the space held by these two arrays
      result = mergeSubArr(recvtmp, subArr, recvtmpLen, subArrLen)
      free recvtmp
      free subArr
      subArr = result
    end if

  if rank 0 process:
    //verify the sorting result
    mpiVerification(serial_sorted_array, mpi_sorted_array)
}
```

# 5 Experimental environment

The experiment is performed in the virtual machine running by VirtualBox. Using Vagrant to set parameters and log in to the virtual machine. Here are the details of the experimental environment:

- **VM Memory**: 4096 MB
- **Number of VM Processor**: 4
- **gcc version**: 7.5.0
- **Operating System**: bento/ubuntu-18.04 powered by VirtualBox
- **Processor**: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz

# 6 Compile & execute method description

Source code compile command: `make -f makefile`

This compiles four programs: `mpisort.c`, `ompmpisort.c`, `serialsort.c`, `ompsort.c`

- `mpisort.c` is designed for random array generation, MPI IO, serial IO and verification, and measuring the elapsed time of sorting array in MPI mode.
- `ompmpisort.c` is designed for sorting array in OMP+MPI mode. The unsorted array is read from the file written by MPI IO module of `mpisort.c`("mpi_unsorted_array.bin").
- `serialsort.c` is designed for measuring the elapsed time of sorting array in serial mode. The unsorted array is also read from "mpi_unsorted_array.bin".
- `ompsort.c` is designed for measuring the elapsed time of sorting array in openmp mode. The unsorted array is also read from "mpi_unsorted_array.bin".

Executing `mpisort`: `mpiexec -n *x* mpisort *algo* *length*`

Executing `ompmpisort`: `mpiexec -n *x* ompmpisort *algo* *length* *y*`

Executing `serialsort`: `./serialsort *algo* *length*`

Executing `ompsort`: `./ompsort *algo* *length* *y*`

*x - number of processes(2/3/4/5...), algo - algorithm parameter(quick/merge/enum/all), length - length of the array to be sorted(1000/10000/200000...), y - number of threads(2/3/4/5...)*

Alternatively, you can try to execute the shell script `./execute.sh`, which will execute all programs for mulitiple times with several number of processes and several number of threads, but it will take a **very long** time. The executing result can be found in `test_data.csv`, which will be introduced later in the speedup analysis section.

**Note**:

1. **Executing mpisort first** before executing the rest programs, as only `mpisort` has the random array generation and file written module, and the other programs need to read the file generated by mpisort in order to get the unsorted array.
2. After executing `mpisort`, **the *length* parameter must be the same value** when excuting the rest programs.

The following image Figure 1 is an example of compiling and executing the source code:

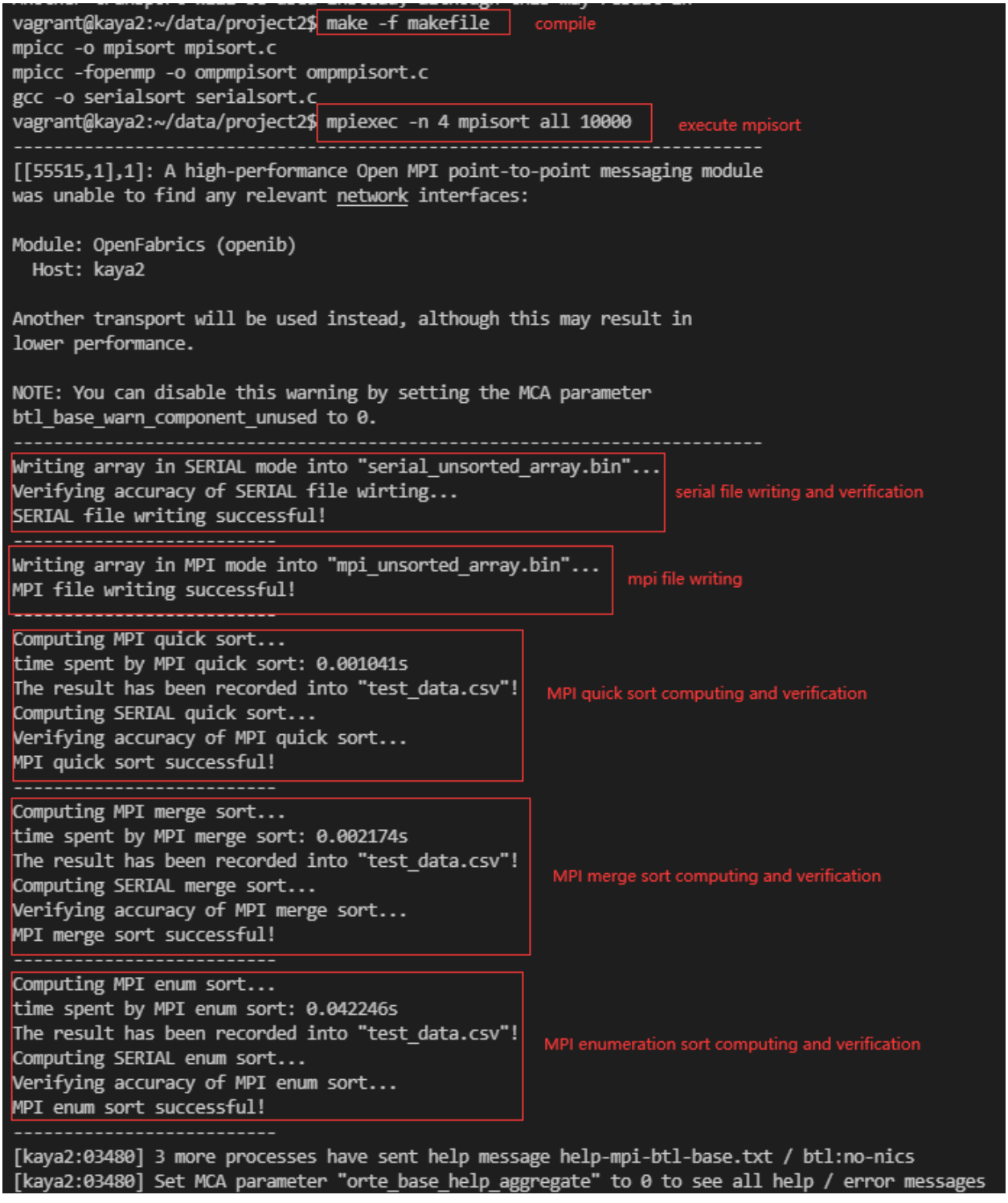



Figure 1: Compiling & Executing

# 7 Speedup analysis

As it was mentioned before, we'll perform the experiment by executing the source code for multiple times with different values of parameters by executing the shell script `execute.sh`.

Here are the detail steps of `execute.sh`:

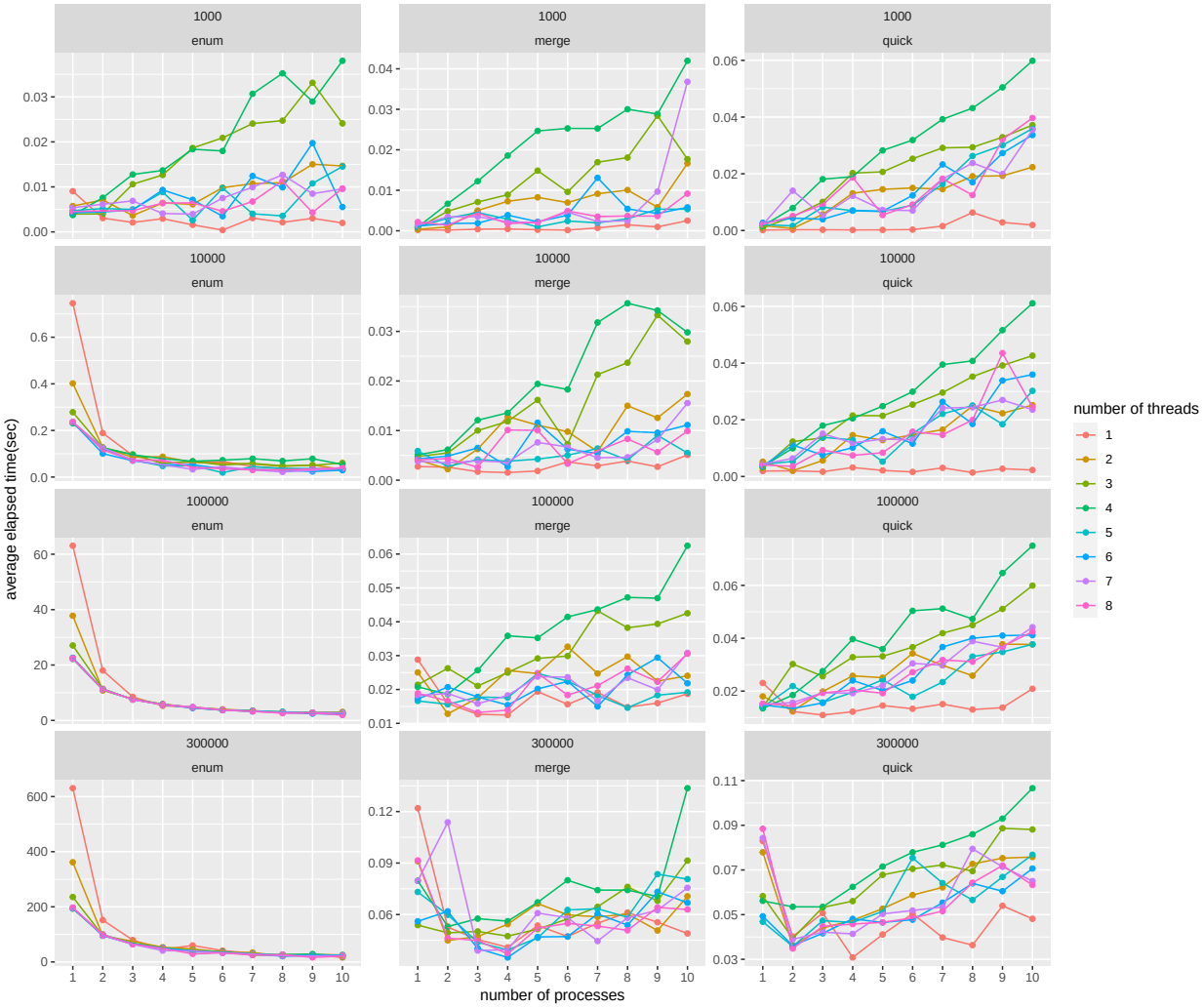
1. we test the length of array in 1k, 10k, 100k, 300k for each sorting algorithms, and 1M, 10M, 100M for quick sort and merge sort only.
2. For each length of the array, we test the number of processes from 1 to 10.
3. For each number of processes, we test the number of threads from 1 to 8.
4. Then we'll do three times for each case in order to have a general result by computing the average elapsed time of each case.

**Note**:

1. In order to avoid the result error due to MPI or OpenMP framework, for cases which the number of processes is 1 and the number of threads is 1, we use `serialsort` to compute the result, and for cases which the number of processes is 1 and the number of threads is from 2 to 8, we use `ompsort` to compute the result.
2. The results of the cases which the number of threads is 1 and the number of processes is from 2 to 10 are computed by executing `mpisort`, and the rest cases are computed by executing `ompmpisort`.

All the testing results are recorded in `test_data.csv`. We'll use ggplot to visualize the `test_data.csv`, which has 4319 observations in total. Here is the source code and images:

```
library("ggplot2")
library("dplyr")
library("gridExtra")
data <- read.csv("test_data.csv")
data1 <- data%>%filter(array_length<=300000)
data1$array_length<-as.factor(data1$array_length)
data1$process_num<-as.factor(data1$process_num)
data1$thread_num<-as.factor(data1$thread_num)
data1<-data1 %>% group_by(array_length, algorithm, process_num, thread_num) %>%
  summarise(average_elapsed_time = mean(time_elapsed))
ggplot(data1,
       aes(x=process_num, y=average_elapsed_time,group=thread_num, color=thread_num))+
  facet_wrap(array_length~algorithm, scales="free_y",ncol=3)+
  geom_point()+geom_line()+labs(x= "number of processes", y="average elapsed time(sec)",
                                color="number of threads")
```



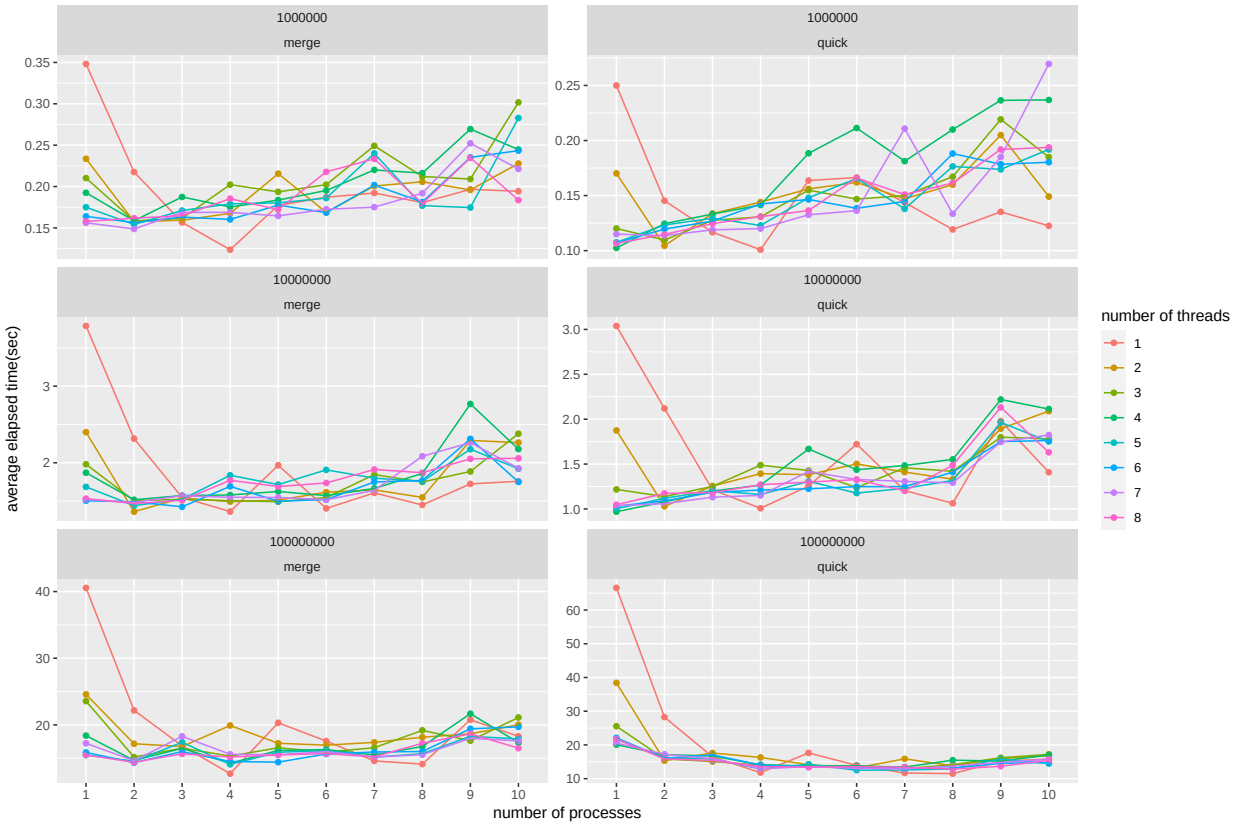

As we can see, the length of array and sorting algorithm are shown in the title of each subplot. For example, the first row is for 1000 array length with each sorting algorithms. The x-axis is the number of processes (hereinafter referred to as NP), and the y-axis is the average elapsed time (hereinafter referred to as AET). Different color of lines represents different number of threads (hereinafter referred to as NT).

For array length at 1k (first row), parallel sorting did not outperform serial sorting for all algorithms, on the contrary, it became much slower when NP and NT increasing (except enumeration sort at NP = 1).

With regard to array length in 10K ~ 300K, parallel sorting performs well in enumeration sort. With the increase of NP, AET shows a general decreasing trend,with less and less slope though, and the difference of AET by different NT become much smaller when NP>2. So the best optimisation of enumeration sort is NP=4 and NT=2, with around 12 times faster than serial sort (NP=1 & NT=1). As for quick sort and merge sort, although it shows some levels of optimisation in NP=2 and different NT when NP=1, parallel sort did not perform well in general due to the small array length.

The following plots shows the performance of quick sort and merge sort with array length in 1M, 10M, 100M.

```
data2 <- data%>%filter(array_length>300000)
data2$array_length<-as.factor(data2$array_length)
data2$process_num<-as.factor(data2$process_num)
data2$thread_num<-as.factor(data2$thread_num)
data2<-data2 %>% group_by(array_length, algorithm, process_num, thread_num) %>%
  summarise(average_elapsed_time = mean(time_elapsed))
ggplot(data2,
       aes(x=process_num, y=average_elapsed_time, group=thread_num, color=thread_num))+
  facet_wrap(array_length~algorithm, scales="free_y",ncol=2)+
  geom_point()+geom_line()+labs(x= "number of processes", y="average elapsed time(sec)",
                                color="number of threads")
```




We can see with the increasing of the array length, the advantage of parallel sorting is becoming more evident. As quick sort and merge sort at 10M and 100M array length show similar speedup trend with enumeration sort at 100K and 300K array length. However, NT still performs well only in NP=1 and NP=2, with 2~4 NT be the most reasonable optimisation.