Parallel Odd-Even Transposition Sort Analysis

SDS 117010008 曹正家

Odd-Even transposition sort is a sorting algorithm commonly used in large clusters due to it's parallel nature. When facing huge amount of data having multiple process running in parallel can sometimes boost the execution speed in a great scale.

In this assignment, I have implemented a parallel odd-even transportation using the MPI library in C. This report is organized as follows: First, I will briefly introduce the algorithm and analyse it's complexity in Section 1. In Section 2, I will introduce my implementation of this program. The experiment results and analysis can be found in Section 3&4. Lastly, I will conclude this program in Section 5.

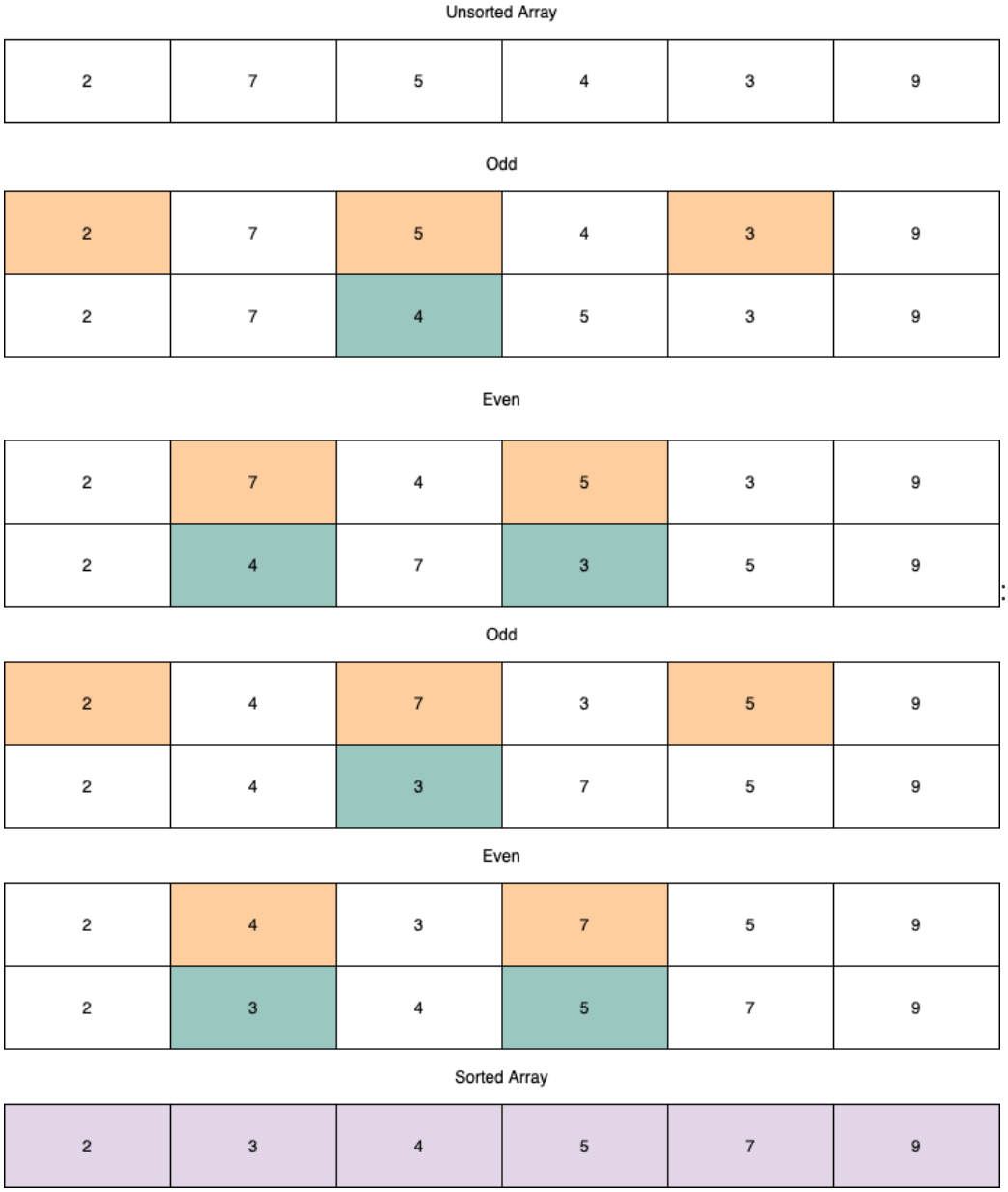
1. Description

In this section, I will briefly introduce how this algorithm works and give a simple complexity analysis of it.

For the sequential Odd-Even transposition sort, the sorting procedure is specified as follows:

> The odd even transposition sort is a variation of bubble sort. Like in bubble sort, elements of a list are compared pairwise and swapped when necessary. However, these compare-swaps are done in two phases: odd and even. Suppose that *a* is a list of integers. The compare-swaps for the phases are as follows:
>
> **Odd phase:** (a[1], a[2]),(a[3], a[4]), (a[5],a[6]), ...
>
> **Even phase:** (a[0], a[1]),(a[2], a[3]), (a[4],a[5]), ...
>
> The algorithm guarantees that for a list with n elements, after n phases the list will be sorted. the list may be sorted with fewer phases, but it will always be sorted after n phases.

The sorting process is illustrated in the following figure:



The yellow block denotes the comparison pointer at each odd/even stage, and the lime blocks denotes a swap action. The original array is unsorted. Then, in the odd phase, the array elements with odd index i will be compared with its adjacent element $i+1$. If the odd element is larger then the neighbor, a swap action is taken. Then in the even phase, array element with even index $i+1$ is compared with the element with index $i+2$. The procedure is repeated recursively until the whole array is sorted.

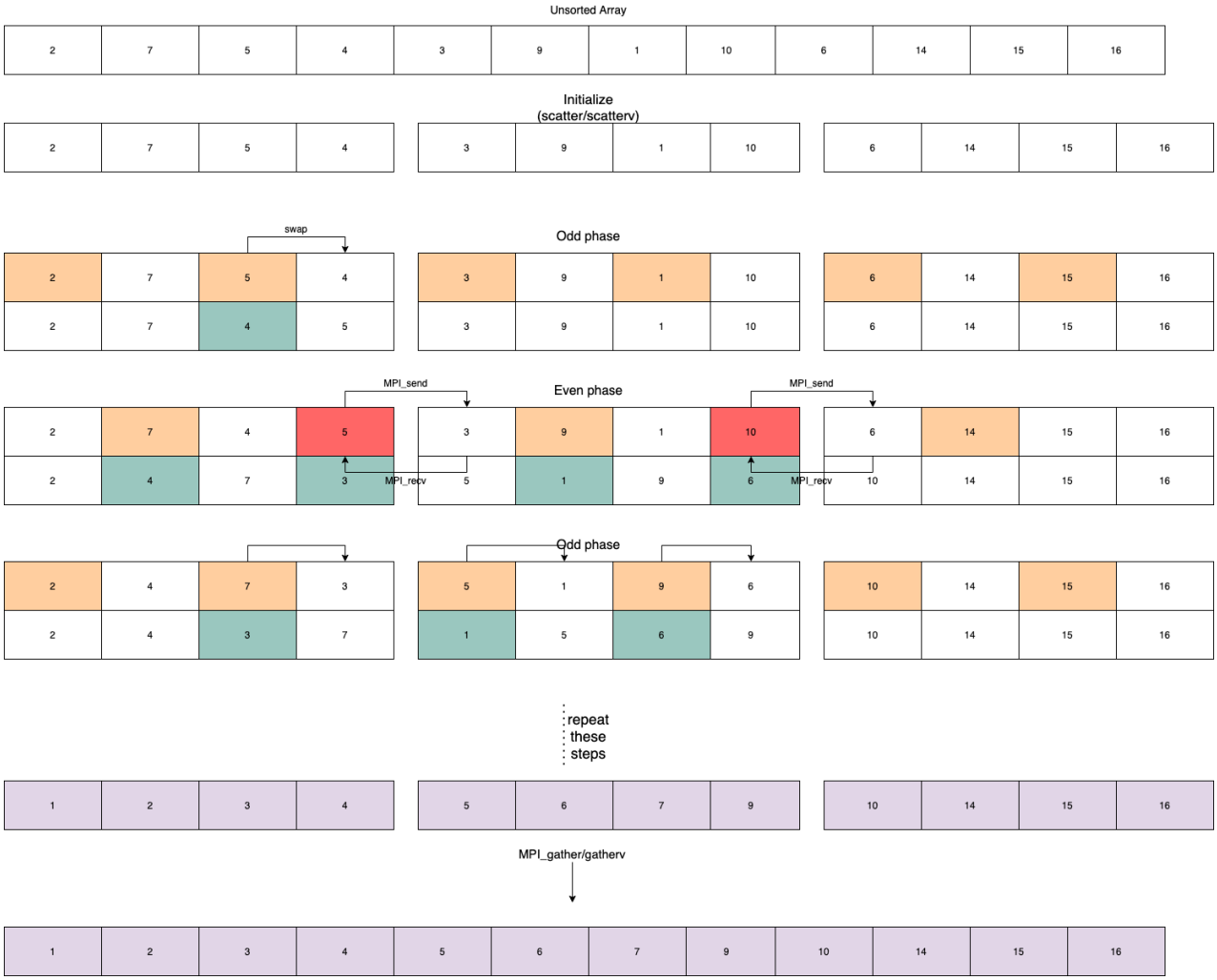
The pseudo code for this sequential algorithm is shown as follows:

```
while (arrayNotSorted) {
  // Odd phase comparison
  if (array[i] < array[i+1]){
    swap(i,i+1)
  }
  // Even phase comparison
  if (array[i+1] < array[i+2]){
    swap(i+1,i+2)
  }
}
```

As we can easily see from the pseudo code here, the sequential odd-even transposition does the sort in $O(n^2)$ time. This is because it does the sorting by iterating over the whole list. It is same as the worst case of quicksort sorting algorithm, and is worse than the merge sort and heap sort, both with worst time complexity of $O(nlog(n))$

- Why do we need transposition sort: Odd-Even in parallel

Although the performance of this sequential sorting algorithm is not the best among all possible candidates, it is still widely used due to the ease of achieving high level of parallelism in the sorting procedure. The parallel sorting is demonstrated in the next picture:



We can see that the original array is now divided into different chunks,the number of which same as the number of processor, each containing array elements to sort. Within each chunk, the odd element with index j will compare with the element j+1 at each odd phase comparison. In the even stage comparison, the element j+1 will be compared with the element j+2. This is the same as the sequential sorting algorithm. The major difference is that when there is a boundary situation, which is the leftover element marked in red, the processes needs to do inter-process communications. In here, we can see that for the process with rank 0 and rank n, it will only need to do one send/recieve pair for the inter-process messages. However, the processes between them need to send/recieve messages both from their previous process and their following process. They would do twice the amount of communication compared with the first and the last process.

The complete complexity analysis of the algorithm is listed here:

Each process can sort in $O(\frac{n}{p}log\frac{n}{p})$, and at each phase (odd/even), the cost of communication is $O(\frac{n}{p})$

The cost of merging two sequence is $O(\frac{n}{p})$, since we have p processes to merge, the total time of communication and merging should be in $O(n)$.

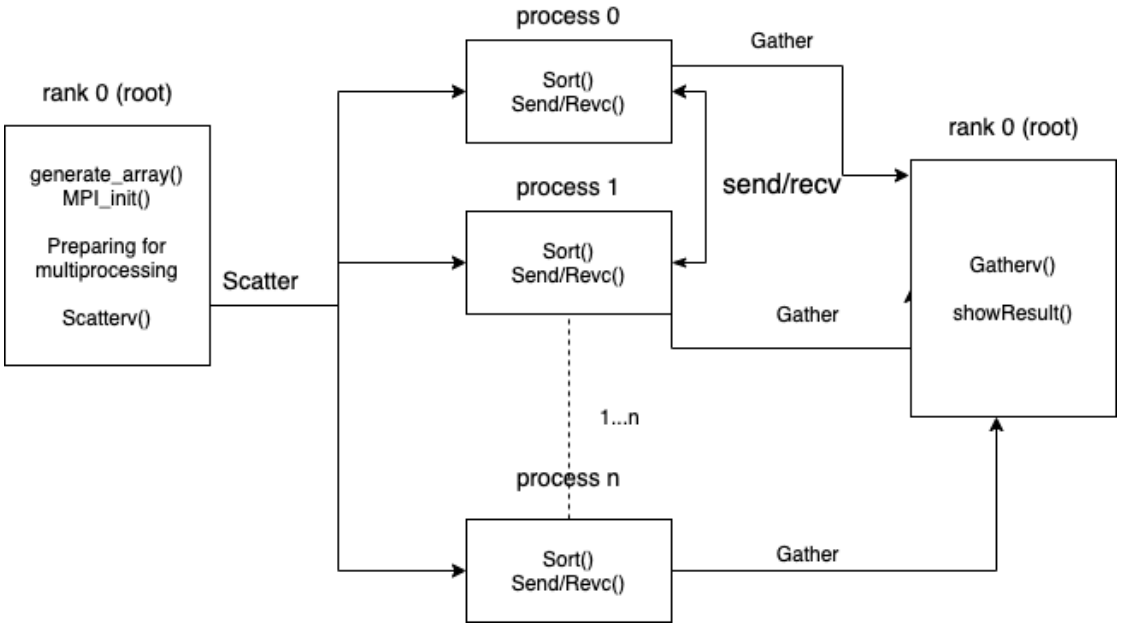
Therefore the total parallel sorting time would be $O(\frac{n}{p}log\frac{n}{p}) + O(n)$

In pseudo code, the parallel odd-even transposition sort can be described as:

```
sortLocalKeys();
for (phase = 0; phase < comm_size; phase++){
  partner_rank = computer_partner(rank)
  if (!idle){
    sendKeysToPartner();
    recvKeysFromPartner();
    if (my_rank < partner_rank){
      keySmallerKeys();
    }
    else{
      keyLargerKeys();
    }
  }
}
```

2. Implementation

The implementation of the sequential sorting program is intuitive. Here we just show the program workflow for the parallel sorting program.



The whole program workflow is shown as the figure. The execution is from left to the right. Roughly, the program can be decomposed to three main parts:

- The first part: Initialization & distribution of array elements.

  the program goes into the main function, it initializes the variables including random array generation function's configuration, the storage used to contain global and local array, their lengths and other information. Then, it does the MPI initialization, including space allocation, specifying communicator, and get the rank & processor number. At the end of this part, the root process called the Scatter function in order to distribute the unsorted array to sub-processes.

- The second part: Local sorting and inter-process message passing

  p processes are going to do the odd even sort together. The MPI_send and MPI_recv function is used to do boundary comparison.

- The third part: Gathering local results and have the sorted array

  he root process call the *MPI_Gather(),* all the sub-processes submit sorted array to the root. Then print the result and record the time used for the sorting algorithm.

In C, the function for sorting is shown as follows:

```
int oddEvenSort(int* localArray,const int chunkSize,const int localRank, const
int globalSize,int pnum, MPI_Comm comm){

  int toSend = 0;
  int toRecv = 0;
    int myRight = (localRank + 1) % pnum;
    int myLeft = (localRank + pnum - 1) % pnum;

    printf("ODD_EVEN_SORT: the array to sort is: \n");
    printArray(localArray,chunkSize);
    printf("\n");
```

```
    for (int i = 0; i < globalSize; i++){
        if (i % 2 == 0){
            for (int j = chunkSize - 1; j > 0; j -= 2) {
                if (localArray[j-1] > localArray[j]){
                    swap(localArray, j, j-1);
                }
            }
        } else {
            for (int j = chunkSize - 2; j > 0; j -= 2) {
                if (localArray[j-1] > localArray[j]){
                    swap(localArray, j, j-1);
                }
            }
            if (localRank != 0){
                toSend = localArray[0];
                MPI_Send(&toSend, 1, MPI_INT, myLeft, 0, comm);
                MPI_Recv(&toRecv, 1, MPI_INT, myLeft, 0, comm,
MPI_STATUS_IGNORE);
                if (toRecv > localArray[0]){
                    localArray[0] = toRecv;
                }
            }
            if (localRank != pnum - 1){
                toSend = localArray[chunkSize-1];
                MPI_Recv(&toRecv, 1, MPI_INT, myRight, 0, comm,
MPI_STATUS_IGNORE);
                MPI_Send(&toSend, 1, MPI_INT, myRight, 0, comm);
                if (toRecv < localArray[chunkSize-1]){
                    localArray[chunkSize-1] = toRecv;
                }
            }
        }
    }

    return 0;
}
```

There is an edge case in the sorting implementation: What if the number of array element cannot be equally spilt into the number of processors? For example, if we have 11 array element and 4 processors, the MPI_Scatter function will do 11/4 = 2 (int constraint), and will send 2 elements to each process. In this way, there are always some leftovers in the array that was not distributed to any process. (The number of the leftover element equals to n%p). To cope with this, we have used the MPI_scatterv and the MPI_gatherv function. The scatterv function can assign different length of array slice to each process. It is the same for gatherv. Now the root process can gather different number of array elements from the working processes.

In my implementation, the method adopted is that I first calculate the $\frac{n}{p}$, and use it as the chunk size for the first n-1 number of processes. Then I calculate the $n\%p$ and add it with the $\frac{n}{p}$, assign the number to the n^{th} process. This is usually faster than spilting the leftover workload to the previous processes and achieve a more balanced structure. The reason is that the leftover number, given that we have a upper bound for process number equals 20, must be smaller than 20. Thus, it will not be too unbalanced and is not worth to maintain a more equal workload to all processes. This is further illustrated in the experiment part.

The code implementation is shown here:

```
if (divisable == 0){
        localArray = (int*) malloc(sizeof(int) * chunkSize);
        MPI_Scatter(globalArray, chunkSize, MPI_INT, localArray, chunkSize,
MPI_INT, 0, comm);
        oddEvenSort(localArray,chunkSize, rank, allSize,pnum, comm);
        MPI_Gather(localArray, chunkSize, MPI_INT, globalArray, chunkSize,
MPI_INT, 0, comm);
    } else {
        int* displs = malloc(sizeof(int) * pnum);
        int* sendCounts = malloc(sizeof(int) * pnum);
        for (int i = 0; i < pnum - 1; i++){
            sendCounts[i] = chunkSize;
        }
        sendCounts[pnum - 1] = chunkSize + divisable;
        displs[0] = 0;
        for (int i = 1; i < pnum; i++){
            displs[i] = displs[i-1] + sendCounts[i-1];
        }
        int recvCount = (rank == pnum - 1) ? chunkSize + divisable : chunkSize;
        localArray = (int*) malloc(sizeof(int) * recvCount);

        MPI_Scatterv(globalArray,sendCounts, displs, MPI_INT, localArray,
recvCount, MPI_INT, 0, comm);
        oddEvenSort(localArray,sendCounts[rank], rank, allSize,pnum, comm);
        MPI_Gatherv(localArray, sendCounts[rank], MPI_INT, globalArray,
sendCounts, displs, MPI_INT, 0, comm);
    }
```

3. Experiments

In this section, we will demonstrate our experiment results and briefly explain its implications. There are mainly two variables in our experiment, the array length and the number of parallel processes.

The local testing environment is:

OS: OS X 10.15.5

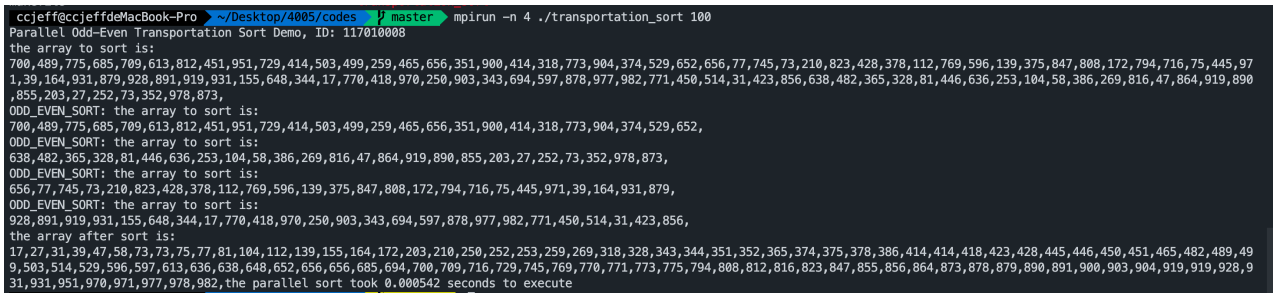
Compiler: Apple clang version 12.0.0 (clang-1200.0.32.2)

MPI: openMPI version 4.2.1

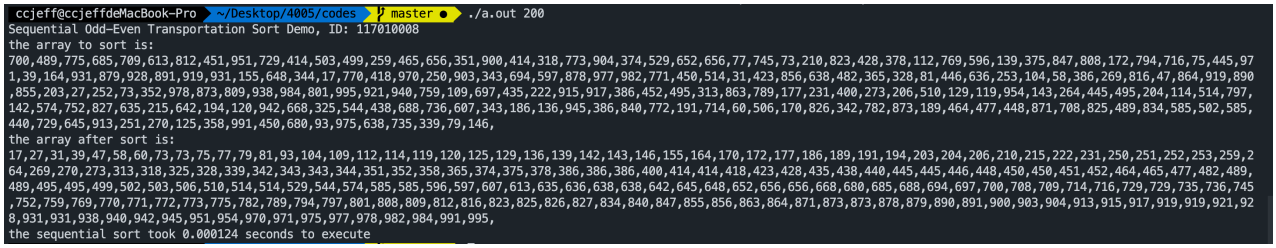
How to run the parallel program in local setup:

1. `mpicc transportation_sort.c -o transportation_sort` or simply use the `make` command to use the make file in the repository.
2. run the program with `mpirun -n $processNumber ./transportation_sort $numberOfArrayElements`

This should give a sample out put like this:



For the sequential program, the procedure is similar. Simply use `clang transportation_sort_seq.c -o seq.out` and then use `./seq $numberOfArrayElements` to run the program. The sample result is demonstrated here:



The testing environment in server is:

```
117010008@master2:~$ cat /proc/cpuinfo | grep "model name"
model name      : Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
model name      : Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
model name      : Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
model name      : Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
model name      : Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
117010008@master2:~$ free -m
              total        used        free      shared  buff/cache   available
Mem:           7880         994        2657          17        4227        6569
Swap:           472           0         472
```

3.1 startup testing

First, we have tested the start up time in the MPI setting by simply clocking the time of MPI_init. The procedue can be illustrated with the pseudo code.

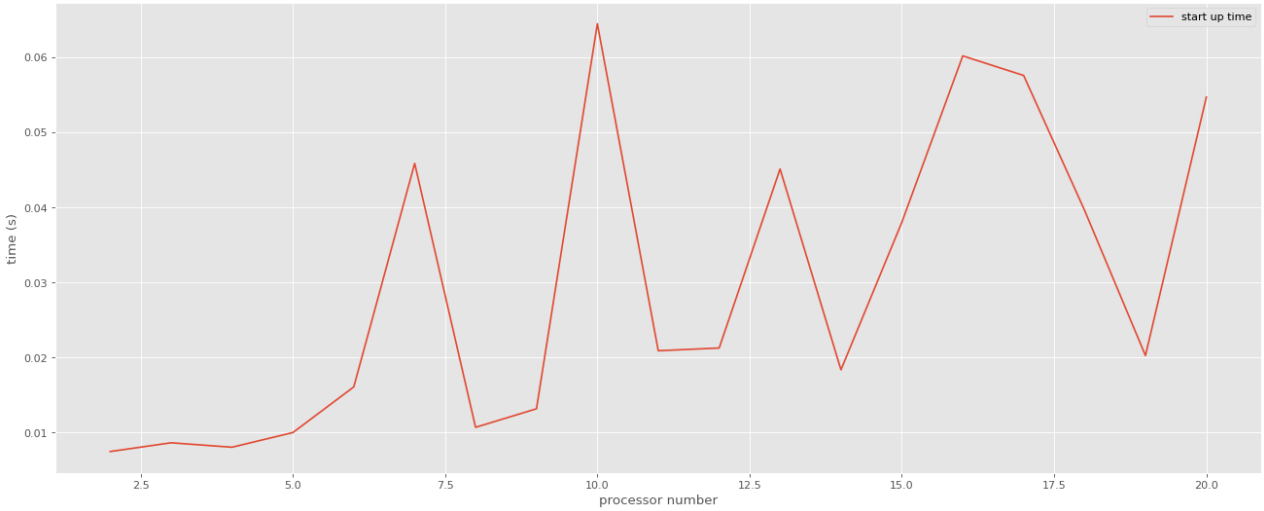
```
//clock here
MPI_Init(&argc, &argv);

comm = MPI_COMM_WORLD;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &pnum);

MPI_Finalize();
//clock here
```

This is to record the start up time of the MPI library, where we have just initialized the processes. Before starting the experiment, we would like to show some properities of the MPI interface during it's startup period. We recorded the time for starting a process with the number of processors given. While there are multiple parallel processing initializing, we recorded the longest start up time in all the processes for each experiment of a fixed processor number. This is because the longest startup time among all the parallel processes is the bottleneck for the system.

The experiment result is shown as here:



Although the trend is not very clear in the figure due to the limited amount of processors given, we can observe that as the number of processor gets larger, the time spent at MPI initialize is longer. The fluctuation at processor number 6, 11, and 17 is most likely to be caused by the inter-node communication through networking during the MPI startup, since on a local node we only have 5 cores to run processes. Thus, we can safely assume that increasing the number of parallel processes will also increase the MPI initialization overhead. Proper number of processes needs to be find out according to each specific problem.

3.2 communication testing

In the second experiment, we record the time spent for each MPI_send()/MPI_recv() pair. This is determined by the communication environment of the system. The ways nodes are arranged have a direct impact on the system performance. We first just send a 0 length message to simply measure the communication overhead of the system, and then analyze if the length of the message will have a direct impact to the system. The experiment is carried out according to the

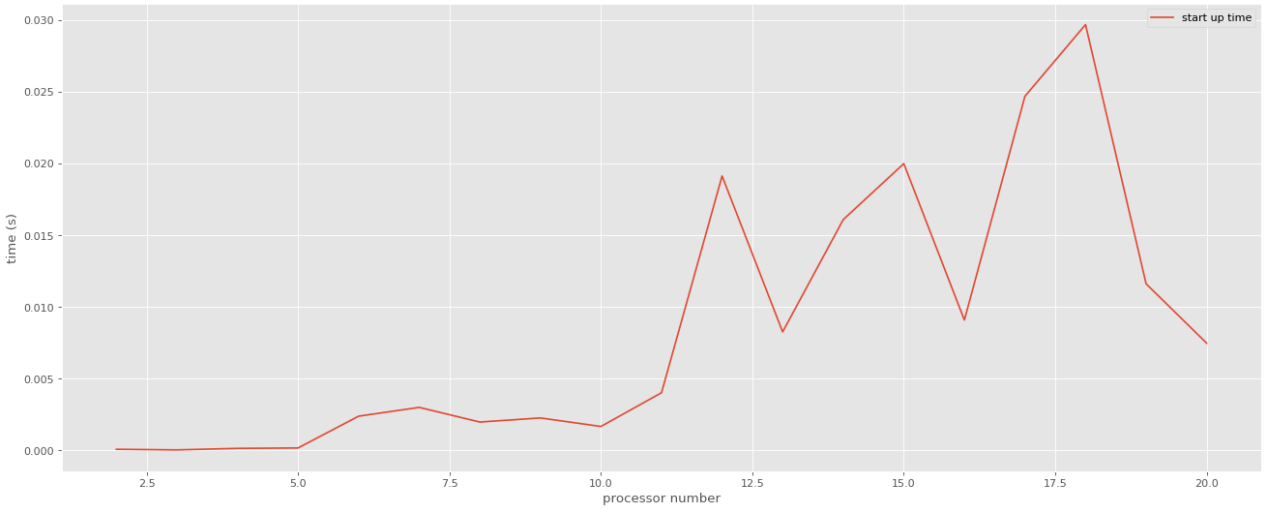
procedures in the pseudo code here:

```
//MPI_init
//clock here
MPI_Send(sendElement, sendSize, MPI_INT, myLeft, 0, comm);
MPI_Recv(recvElement, recvSize, MPI_INT, myLeft, 0, comm, MPI_STATUS_IGNORE);
//clock here
//MPI_Finalize
```

communication startup and network cost

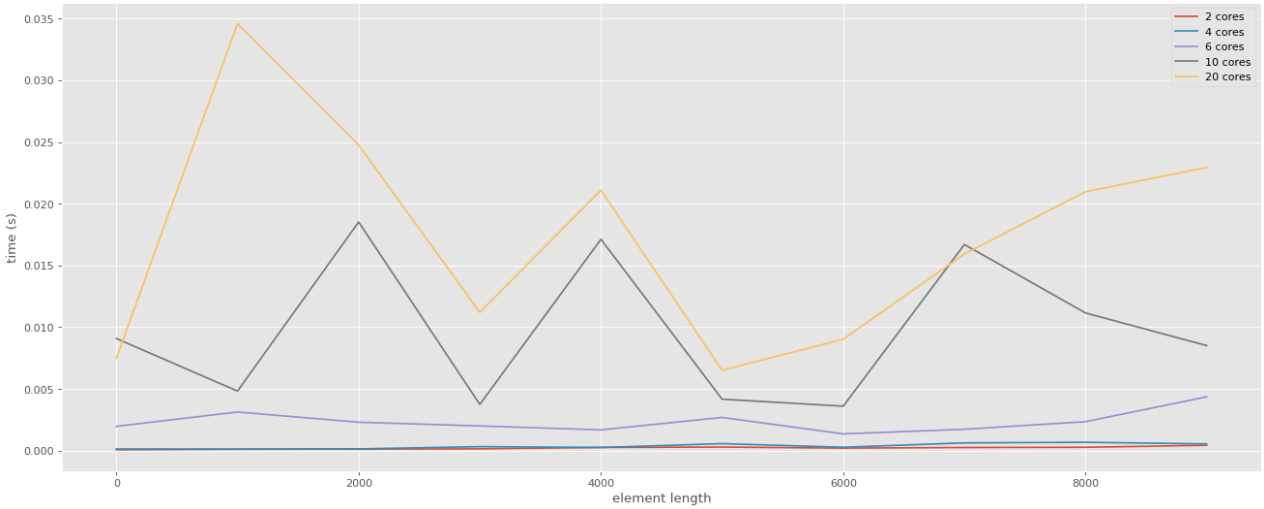
There are again two main factors affecting the communication process in this program: the processor number and the element sent. Although it is clear that in this transposition sort algorithm we only need to send 1 array element for each send/recv pair, the characteristics of send a large array of elements in MPI network communication is still worth exploring since it reveals if we should use the transposition sort method by equally dividing the complete array to each processes using the MPI_scatterv and MPI_gatherv. These two operations involve sending a variable length of array in the initializing stage. The characteristics of the communication system are shown in the following figures.

Here shows the performance of each processors sending an empty array, which simply measures the communication overhead at each processor.



As we can see from the figure, apart from some fluctuations in the network, the general trend is that as the number of processes increases, the communication overhead also increases. This effect is more obvious in the next figure.

Then, we would like to see if the increase in the array size will have a direct impact on the MPI communication. Here we fix the process number and only change the communication size. For simplicity, I have only drew 5 lines each depicting 2,4,8,16,20 processes communicating by sending varing length array in parallel.

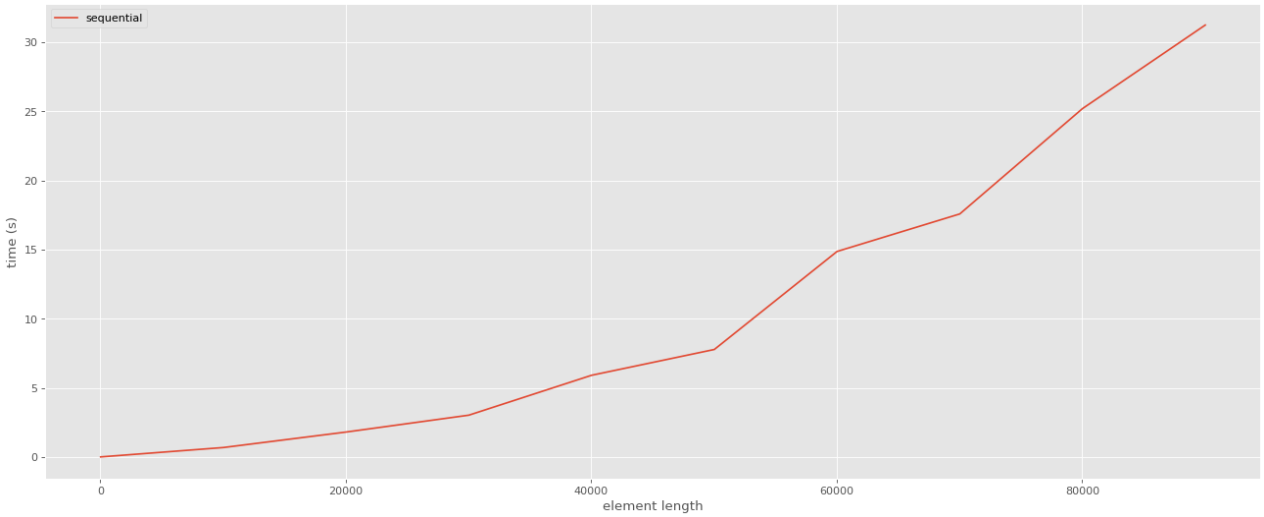


It can be easily seen that as the array element length varies, there is not an obvious change in the communication time. Rather, the communication overhead is more obvious in the figure here. For example, there is a large overhead difference between the inter-20-processes communicaiton line and the inter-4-processes communication time. Thus, we can see that the communication overhead is more deterministic to the overall communication cost compared to the communication message length.

The experiments show that the major cost of communication is in the communication startup stage. In our program, the main bottleneck during the communication stage is the communication overhead. This implies that during the sort, rather than controlling the size of an array, we should consider eliminating the number of send/recv pair during the sort to achieve better performance.

3.3 sequential execution

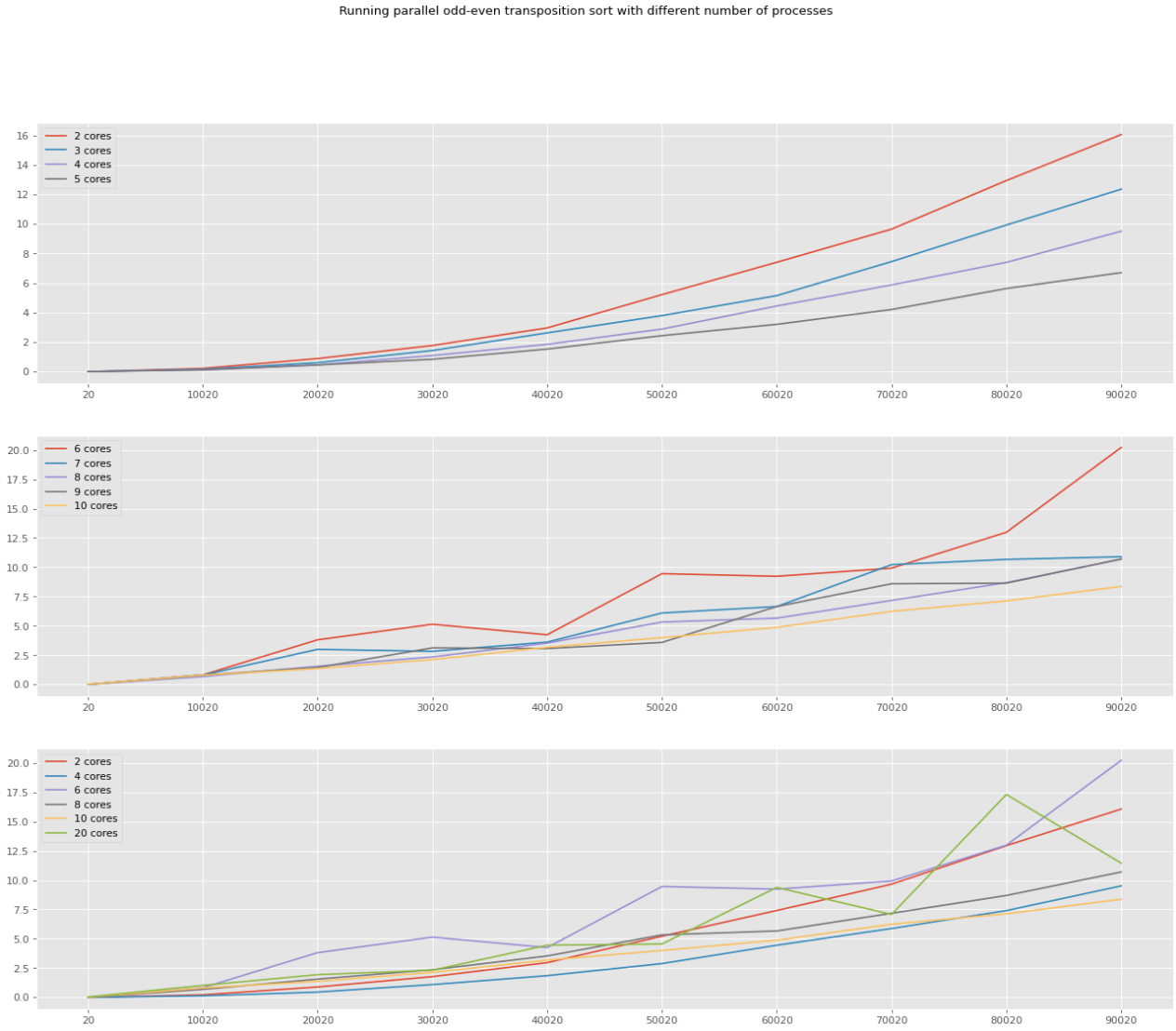
As we have illustrated before in section 1, the sequential odd-even transposition sort should run in $O(n^2)$ for average case, and $O(n^2)$ in worst case. Here we demonstrate the complexity of the algorithm with the experiment figure below:



We can easily see from the picture here that the time grows in $O(n^2)$ order in time complexity.

3.4 parallel execution

We then measures the program execution in parallel to see it's characteristics. Due to the inter-node communication, there are some fluctuation due to the network latency. However, as the number of processes increases, it also increases the level of parallelism, which results in a decreased sorting time.



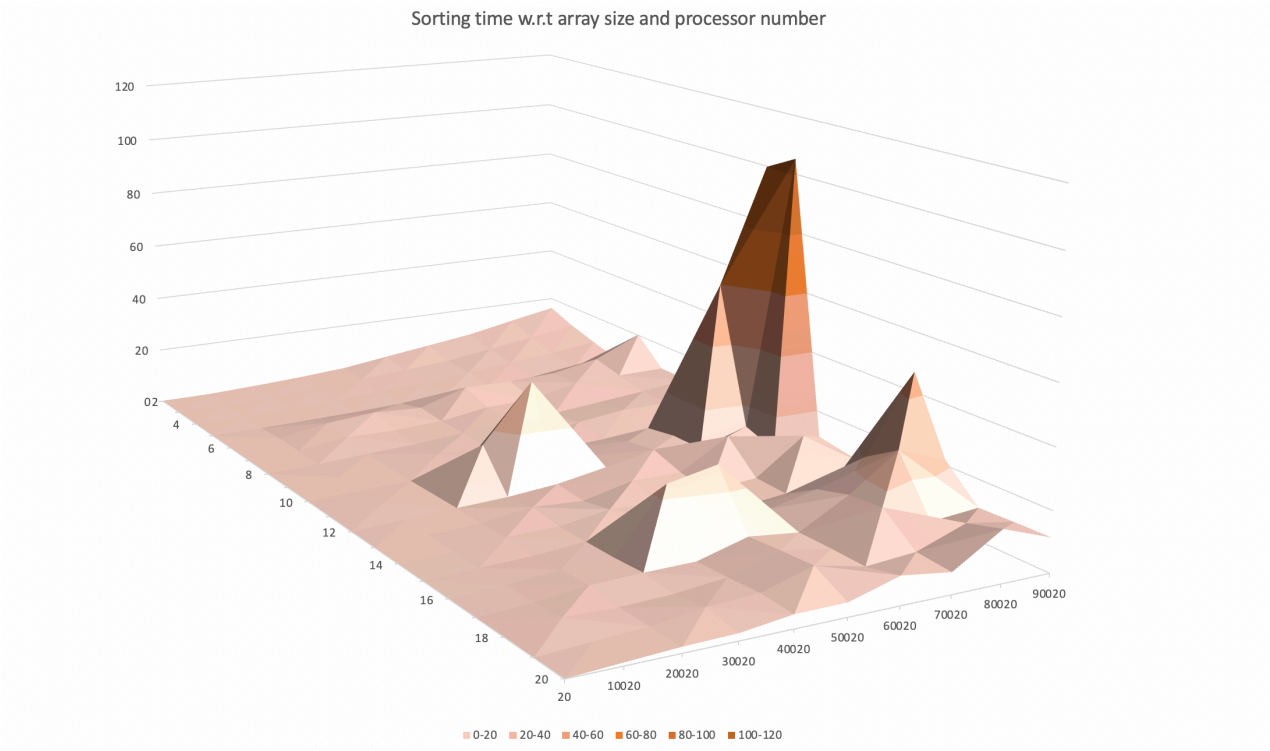
It is clear that the parallel sort can do the computation in close to time order $O(n^2)$ (= $O(nlogn)$, to be precise).

From the first subgraph, we can see that as the number of processes increases, the sorting time decreases. When we have the process number smaller or equals to 5, there are only intra-node communications. That is, the processes are running on different cores of a single CPU. This communication is much faster than internode communication. At this situation, we can see that if we increase the number of processes, we can imporve the overall performance.

In the second subgraph, we see that if we have introduced the 6^{th} process, the performance is the worst among all processes. This is because there involves interprocess communication. Again, if we increase the process number, the number require for sorting decreases due to a higher level of parallelism.

However, just like we have stated before, increase the number of processes will also cause the increase in processes initialization overhead and communication overhead. As shown in the third subfigure, the optimal number of processes here is not the largest one 20, rather it is 4 processes for smaller arrays and 10 processes for larger arrays. This is a trade-off between the parallelism speedup and the parallel overhead.

To have a complete overview of the experimental data, I have included the 3D graph, with x axis the process number, y axis the array size, and z axis the sorting time for seeing the optimal solution.



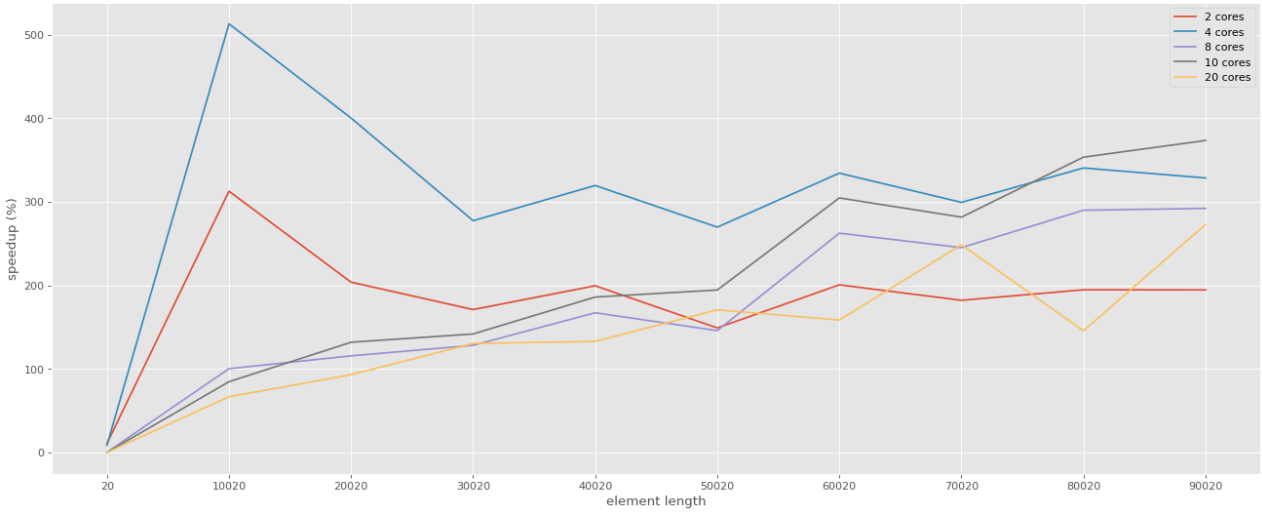
4. Analysis

In this section, we compare the parallel algorithm with the sequential algorithm to have a better understanding of the parallel speed up in sorting.

According to Amdahel's law, we define speed-up value as: $Speedup = \frac{sequentialTime}{parallelTime}$.

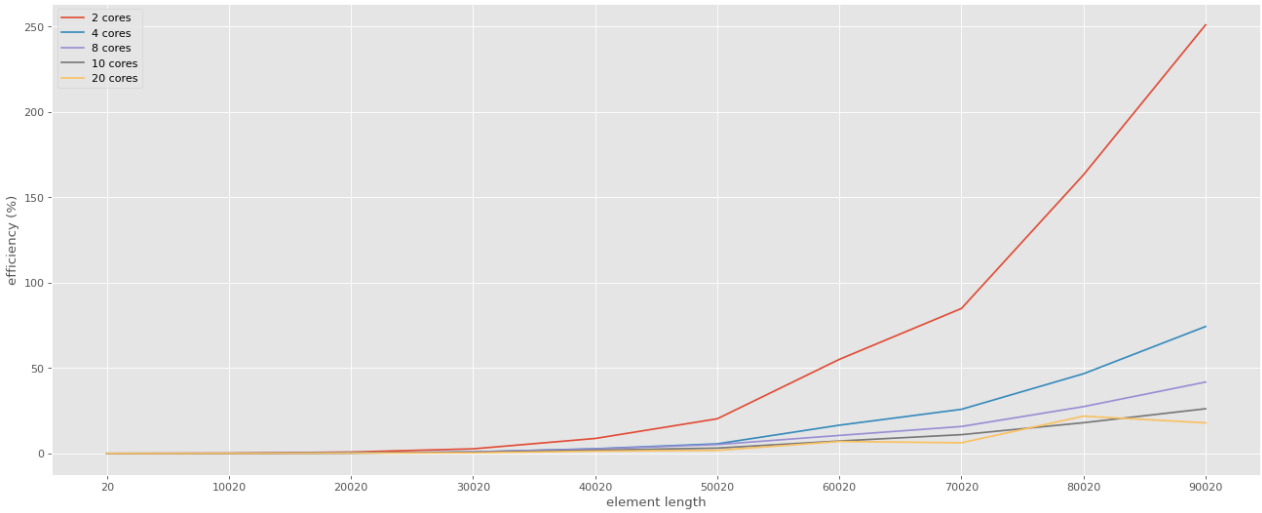
The efficiency of the program is defined as $Efficiency = \frac{sequentialTime}{p*parallelTime}$

With the formula, we calculate the speed up of different processors when sorting different array:



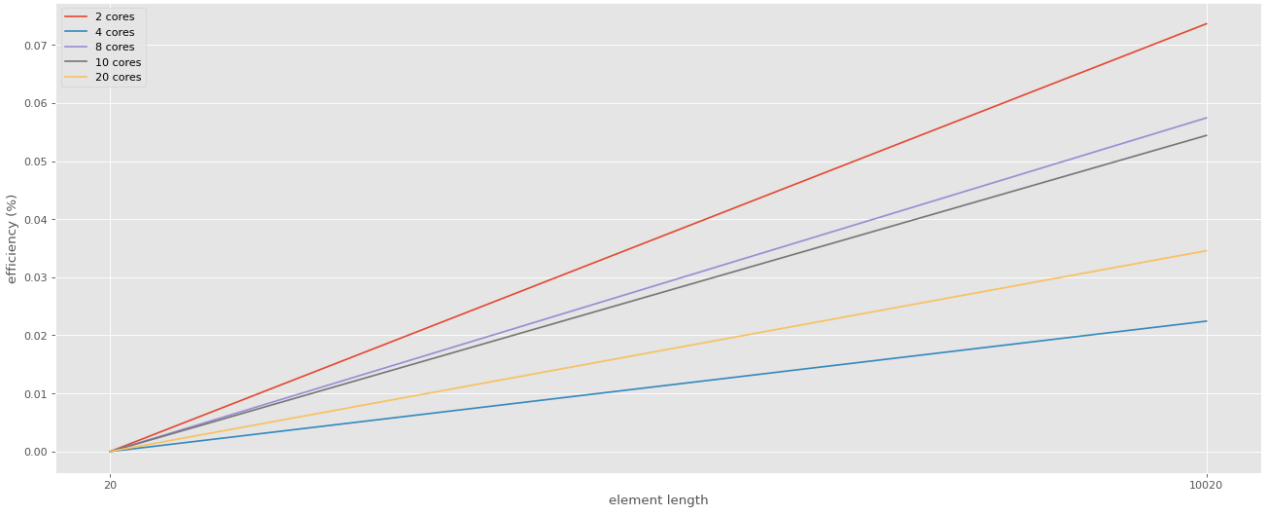
Due to the run time constraint, I have only tested the program with maximum array size 90020, where we see the speedup of 10 cores surpassing 4 cores at array size close to 70020. It means that more process will only bring more speedup only when the array size is large enough. In general, despite the large network overhead and initialization overhead, the parallel program generally achieves a satisfying level of speedup comparing to the sequential version.

However, having a speedup does not mean it is always efficient to use the parallel sorting algorithm. In terms of efficiency, it is not always worth to use more processes. The following figure plots the efficiency factor of different processors when dealing with different arrays:



Thus, we can see that when the array size is very small, the parallel program will not bring the improvement in terms of efficiency. It is very close to 0, which means that the sequential program is actually better than the parallel program. When the array size becomes larger, there began to be an increase in terms of efficiency when using parallel program. From the above diagram, we can tell that despite sorting with 4 parallel processes give the largest speedup, it is more efficient to sort with 2 processes in parallel.

If we zoom in to the small-array size region, we can find the following graph:



This illustrates that when the array size is small, the larger process number actually is actually the same efficient as the smaller ones. However, as the size increases, it drops faster than the parallel program executing with smaller process number.

5. Conclusion & Future works

In this project, sequential and parallel odd even sorting is implemented to conduct experiments on their performance. I have tested the startup time for MPI_initialization and MPI_communication. It is illustrated that the speedup overhead grows if there are more processes running in parallel. From the experiments , I discovered that when the process number is increased, the parallel program does give a faster sorting. However, by comparing with the sequential execution time, we see that the speedup is not always the largest when there are more processes running in parallel. Moreover, in terms of efficiency, it is not always the most cost-effective method to simply add the process number. This is most likely be caused by the MPI overheads.

Due to the hardware constraints, some of the experiments were not carried out. This will be discussed in future:

- should add the array size to a larger number to see if 20 processes will eventually surpass all other methods.
- should compare multi-processing with multithreading and distinguish their use case and speedup factor.

Reference

1. Odd Even Transposition Sort

   http://selkie-macalester.org/csinparallel/modules/MPIProgramming/build/html/oddEvenSort/oddEven.html#merge-low-and-merge-high

2. MPI_scatterv

   https://mpi.deino.net/mpi_functions/MPI_Scatterv.html

3. Parallel speedup

https://software.intel.com/content/www/us/en/develop/articles/predicting-and-measuring-parallel-performance.html#:~:text=Simply%20stated%2C%20speedup%20is%20the,6720%2F126.7%20%3D%2053.038).

Appendix

Sequential program in full code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//helper functions
int generateArray(int* target,int size){
    srand(100);   // Initialization
    for (int i = 0; i < size; i++){
        target[i] = rand()%1000;
        // printf("the number of index %d is: %d \n",i,target[i]);
    }
    return 0;
}

int printArray(int *target, int size){
    for (int i = 0; i < size; i++){
        printf("%d,",target[i]);
    }
    printf("\n");
    return 0;
}

int swap(int* array, int startPos, int endPos)
{
    int temp = array[startPos];
    array[startPos] = array[endPos];
    array[endPos] = temp;
    return 0;
}

int oddEvenSort(int* arr, int n){
    int isSorted = 0;

    while(!isSorted){
        isSorted = 1;
        for(int i = 1; i <= n-2; i = i+2){
            if(arr[i] > arr[i+1]){
                swap(arr, i, i+1);
                isSorted = 0;
            }
            if(arr[n-1] < arr[n-2]){
```

```
                swap(arr, n-1, n-2);
                isSorted = 0;
            }
        }

        for(int i = 0; i < n-2; i = i+2){
            if(arr[i] > arr[i+1]){
                swap(arr, i, i+1);
                isSorted = 0;
            }
        }

    }

    return isSorted;
}


int main(int argc, char** argv){
    if (argc != 2){
        return 0;
    }
    int allSize = atoi(argv[1]);
    int* allArray = malloc(sizeof(int) * allSize);
    generateArray(allArray,allSize);
    printf("Sequential Odd-Even Transportation Sort Demo, ID: %d \n",
117010008);
    printf("the array to sort is: \n");
    printArray(allArray,allSize);

    clock_t startTime, endTime,duration;
    startTime = clock();
    oddEvenSort(allArray, allSize);
    endTime = clock();
    duration = endTime - startTime;
    double time_taken = ((double)duration)/CLOCKS_PER_SEC; // in seconds

    printf("the array after sort is: \n");
    printArray(allArray,allSize);

    printf("the sequential sort took %f seconds to execute \n", time_taken);

    return 0;
}
```

Parallel version with MPI:

```
#include "mpi.h"
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//helper functions
int generateArray(int* target,int size){
    srand(100);   // Initialization
    for (int i = 0; i < size; i++){
        target[i] = rand()%1000;
        // printf("the number of index %d is: %d \n",i,target[i]);
    }
    return 0;
}

int printArray(int *target, int size){
    for (int i = 0; i < size; i++){
        printf("%d,",target[i]);
    }
    return 0;
}

int swap(int* array, int startPos, int endPos)
{
    int temp = array[startPos];
    array[startPos] = array[endPos];
    array[endPos] = temp;

    return 0;
}


int oddEvenSort(int* localArray,const int chunkSize,const int localRank, const
int globalSize,int pnum, MPI_Comm comm){

  int toSend = 0;
  int toRecv = 0;
    int myRight = (localRank + 1) % pnum;
    int myLeft = (localRank + pnum - 1) % pnum;

    printf("ODD_EVEN_SORT: the array to sort is: \n");
    printArray(localArray,chunkSize);
    printf("\n");

    for (int i = 0; i < globalSize; i++){
        if (i % 2 == 0){
            for (int j = chunkSize - 1; j > 0; j -= 2) {
                if (localArray[j-1] > localArray[j]){
                    swap(localArray, j, j-1);
                }
```

```
            }
        } else {
            for (int j = chunkSize - 2; j > 0; j -= 2) {
                if (localArray[j-1] > localArray[j]){
                    swap(localArray, j, j-1);
                }
            }
            if (localRank != 0){
                toSend = localArray[0];
        MPI_Send(&toSend, 1, MPI_INT, myLeft, 0, comm);
        MPI_Recv(&toRecv, 1, MPI_INT, myLeft, 0, comm, MPI_STATUS_IGNORE);
        if (toRecv > localArray[0]){
                    localArray[0] = toRecv;
                }
            }
            if (localRank != pnum - 1){
                toSend = localArray[chunkSize-1];
                MPI_Recv(&toRecv, 1, MPI_INT, myRight, 0, comm,
MPI_STATUS_IGNORE);
        MPI_Send(&toSend, 1, MPI_INT, myRight, 0, comm);
                if (toRecv < localArray[chunkSize-1]){
                    localArray[chunkSize-1] = toRecv;
                }
            }
        }
    }

    return 0;
}

int main(int argc, char** argv){
    // int* unsorted = generateArray(1000);
    if (argc != 2){
        printf("Please enter the array size in command line arguments!\n");
        return 0;
    }
    MPI_Comm comm;
  int rank;
  int pnum;
  int allSize = atoi(argv[1]);
  int chunkSize = 0;
    int isOdd;

    MPI_Init(&argc, &argv);

  comm = MPI_COMM_WORLD;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &pnum);
```

```
    int* globalArray = (int*) malloc(sizeof(int) * allSize);

    //divisability check
    int divisable = allSize%pnum;
  chunkSize = allSize / pnum;

    if (rank == 0){
        generateArray(globalArray,allSize);
        printf("Parallel Odd-Even Transportation Sort Demo, ID: %d \n",
117010008);
        printf("the array to sort is: \n");
        printArray(globalArray,allSize);
        printf("\n");
    }
    // int* localArray = (int*) malloc(sizeof(int) * chunkSize);
    int* localArray;

    clock_t startTime, endTime,duration;
    startTime = clock();
    if (divisable == 0){
        localArray = (int*) malloc(sizeof(int) * chunkSize);
        MPI_Scatter(globalArray, chunkSize, MPI_INT, localArray, chunkSize,
MPI_INT, 0, comm);
      oddEvenSort(localArray,chunkSize, rank, allSize,pnum, comm);
      MPI_Gather(localArray, chunkSize, MPI_INT, globalArray, chunkSize,
MPI_INT, 0, comm);
    } else {
        int* displs = malloc(sizeof(int) * pnum);
        int* sendCounts = malloc(sizeof(int) * pnum);
        for (int i = 0; i < pnum - 1; i++){
            sendCounts[i] = chunkSize;
        }
        sendCounts[pnum - 1] = chunkSize + divisable;
        displs[0] = 0;
        for (int i = 1; i < pnum; i++){
            displs[i] = displs[i-1] + sendCounts[i-1];
        }
        if (rank == 0){
            printf("aaa\n");
            printArray(displs,pnum);
            printf("aaa\n");
            printArray(sendCounts,pnum);
            printf("aaa\n");

        }

        int recvCount = (rank == pnum - 1) ? chunkSize + divisable : chunkSize;
        localArray = (int*) malloc(sizeof(int) * recvCount);
```

```
        MPI_Scatterv(globalArray,sendCounts, displs, MPI_INT, localArray,
recvCount, MPI_INT, 0, comm);
        oddEvenSort(localArray,sendCounts[rank], rank, allSize,pnum, comm);
        MPI_Gatherv(localArray, sendCounts[rank], MPI_INT, globalArray,
sendCounts, displs, MPI_INT, 0, comm);

        // MPI_Scatterv(globalArray, sendCount,0, MPI_INT, localArray,
recvCount, MPI_INT, 0, comm);
        // printf("sendCount for rank %d is: %d \n", rank, sendCount);
    }

    endTime = clock();
    duration = endTime - startTime;
    double time_taken = ((double)duration)/CLOCKS_PER_SEC; // in seconds

    if (rank == 0){
        printf("the array after sort is: \n");
        printArray(globalArray,allSize);
        printf("the parallel sort took %f seconds to execute \n", time_taken);
    }
    free(localArray);

    MPI_Finalize();

    free(globalArray);

    return 0;
}
```

Every other related material can be find in: https://github.com/ccjeff/CSC4005_parallel_computing