Parallel Odd-Even Transposition Sort Analysis

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.

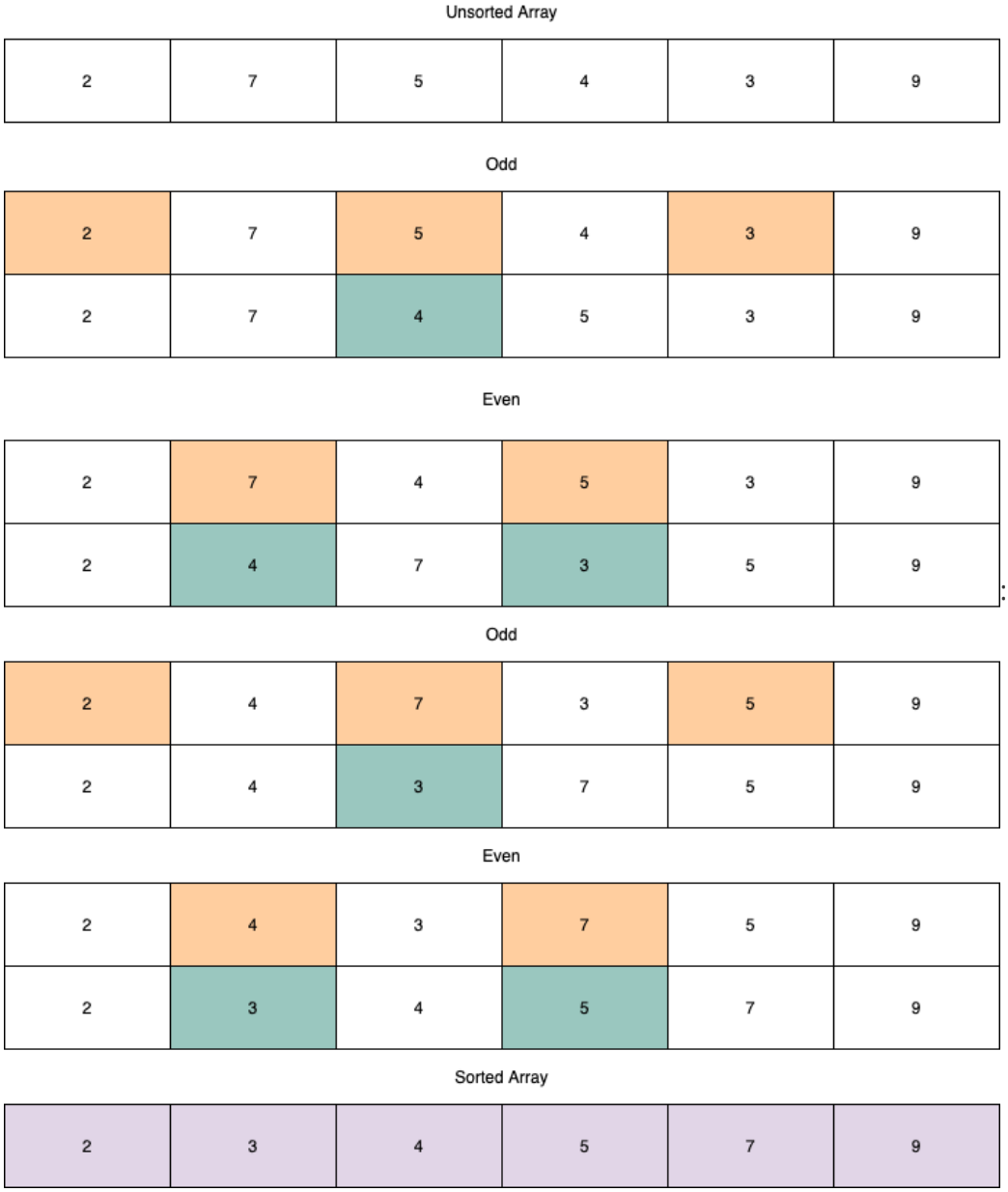
1. Description

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.

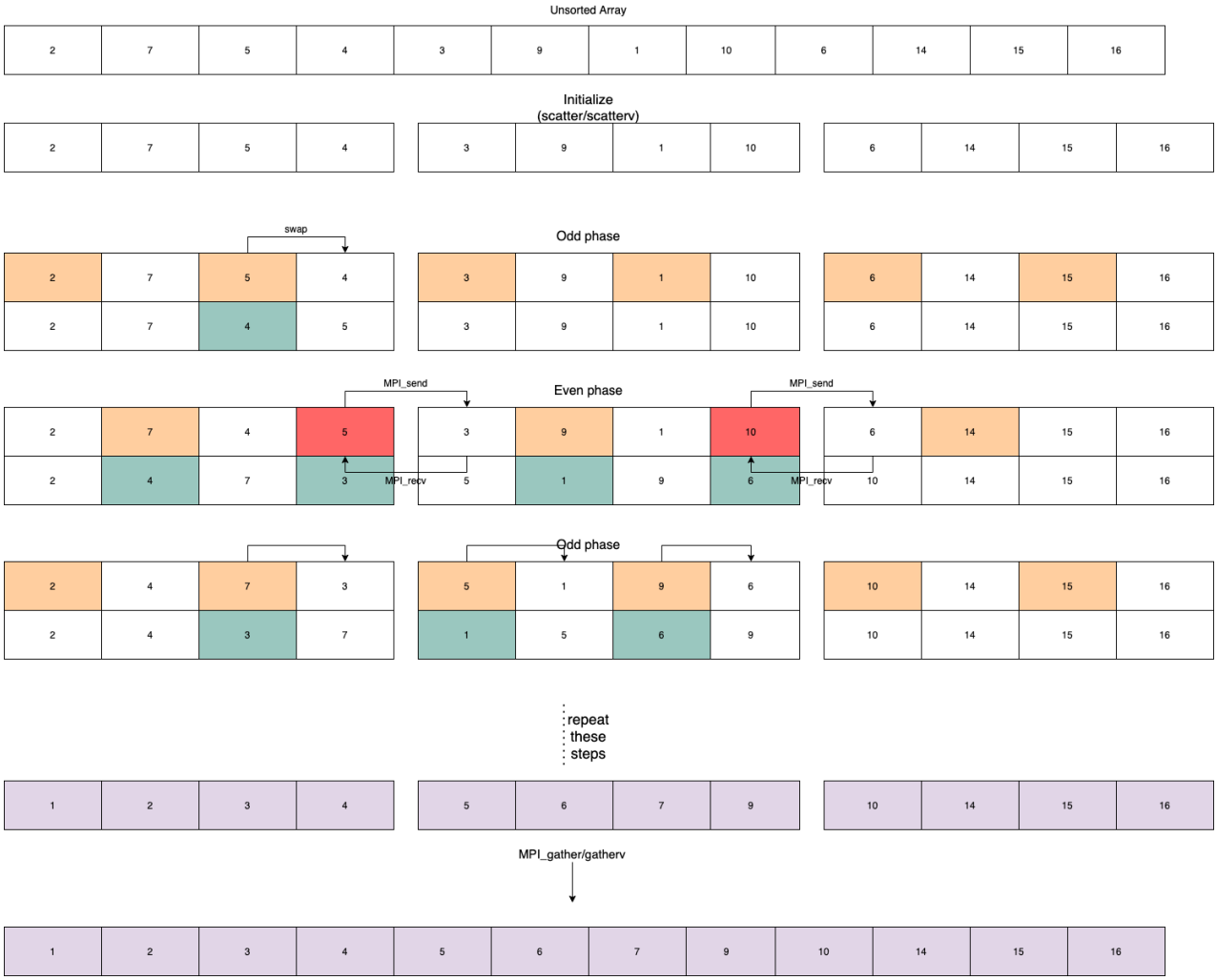
The pseudo code for this sequential algorithm is shown as follows:

```
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 $O(n)$.

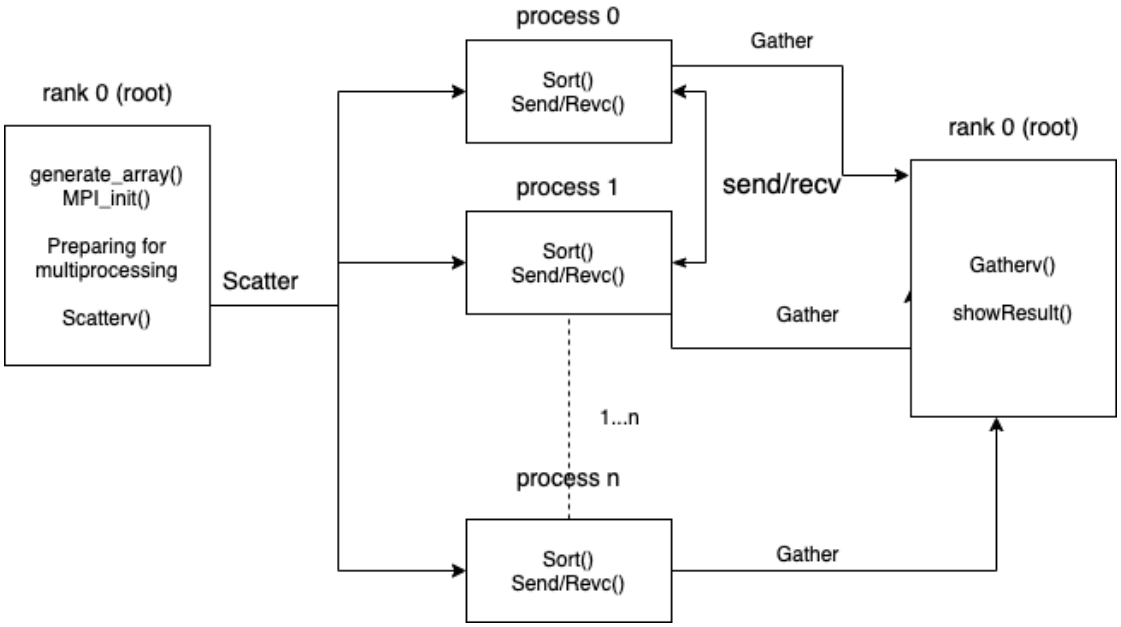
Therefore the total parallel sorting time would be $O(\frac{n}{p}log\frac{n}{p}) + O(n)$

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 program in local se

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
```

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