CSC6220 Final Project

Anna Paula Pawlicka Maule - gl0779

I. INTRODUCTION

The goal of this final project is to implement a Shell-Sort algorithm in parallel and compare it's performance with a serial Quick Sort, and Odd-Even Sort in parallel. The testing cases are arrays of size $2^{16}, 2^{20}, 2^{24}$, and 2^{30}.

II. METHOD

- The first phase of the Shell-Sort algorithm was based on the pseudo code provided on the project's specification Figure 1. The Second phase it was modified. I used a MPI_AllReduce() to check if there were any swaps.

Algorithm 1 Shell-sort like parallel algorithm

```
1: {Phase I: Hypercube Compare-Exchange.}
2: for i = (d − 1) to 0 do
3:    if (i-th bit of rank) = 1 then
4:       compare-split-hi(i);
5:    else
6:       compare-split-low(i);
```

Fig. 1. Pseudo Code

- On the Shell-Sort Algorithm I used MPI_Send() and MPI_Recv() instead of MPI_SendRecv(). The implementation of the processor's mapping was implemented as described on the assignment. Follows an illustration of how the mapping was done by displaying the processors communication/exchange on Figure 2.

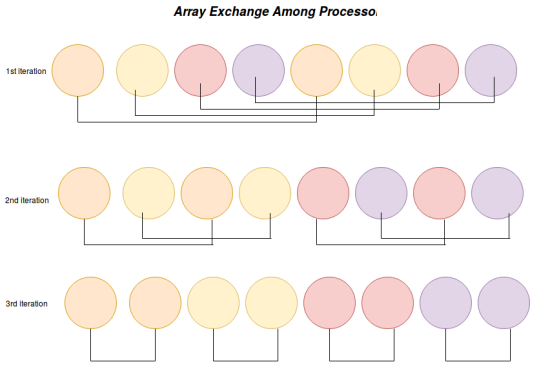

Fig. 2. Processors Communication Phase 1 Shell-Sort

- For the compare and split function. I also implemented in a more optimal way. Where, instead of merging the two arrays and then splitting it in half keeping only the half that matters, my implementation checks what half will be kept and selects the elements from the two arrays that will compose the final array in n steps. Instead of 2n, where n is the number of elements in each array. The pseudo-code for that function is on Figure 3.

```
if(keepSmallValues)
        k=j=0;
        for(i=0; i<size; i++)
                if(localArray[k] <= recvArray[j])
                        finalArray[i] = localArray[k];
                        k++;
                else
                        finalArray[i] = recvArray[j];
                        j++;
else
        k=j=size-1;
        for(i=size-1; i>=0; i--)
                if(localArray[k] => recvArray[j])
                        finalArray[i] = localArray[k];
                        k--;
                else
                        finalArray[i] = recvArray[j];
                        j--;
return finalArray
```

Fig. 3. Pseudo Code

III. RESULTS

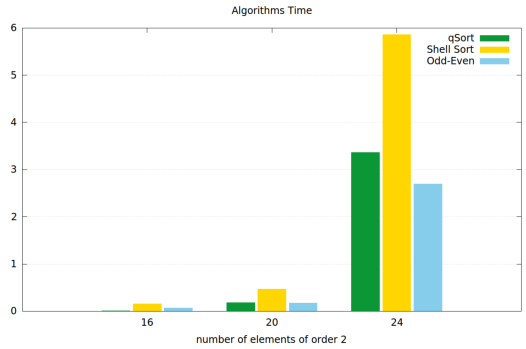

Fig. 4. Algorithms times, y columns is an unit of time

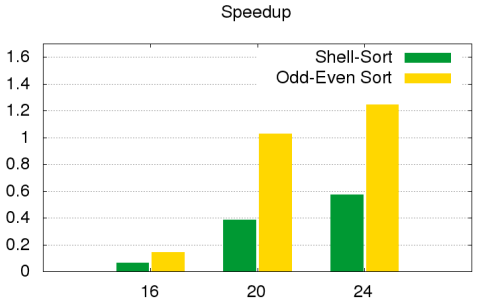

Fig. 5. Speedup

From the histogram above (Figure 4) and speedup plot(Firgure 5), it is possible to conclude that for 2^{16}

elements, the serial implementation of quick sort is faster than Shell-Sort and Odd-Even Sort in parallel.

For 2^{20} elements the odd-even outperforms the quick sort, it is not very noticeable on the plot. However, for 2^{24} elements it becomes more visible Odd-Even's better performance. The Shell sorting is the slowest, and that it is more noticeable when there is 2^{24} elements too. The reason why Shell-Sort is getting a worse performance is due the way the stop criteria was implemented. To verify if there is no more swaps, there is a variable that keeps track of it in each processor. However, to check if there was not any swap in any process, I used a MPI_AllReduce() function to sum the amount of swaps in each processor. AllReduce() waits for all process, so that makes that section of the code serialized, and there is on top of that the overhead of exchanging the swap count among all processors.

Another factor is the difference in number of phases between Odd-Even and Shell-Sort. Odd-Even always performs 8 phases, to guarantee the worst case scenario, where the smallest element is in the last processor or the biggest element is in the first processor travels to its right position. The Shell-sort performs by default 3 phases ($\log 8$ and at least 2 phases of the odd-even phase. So in the best case scenario of this implementation the shell-sort will perform 3 phases less than the odd-even. However, the barrier created by the MPI_AllReduce() and the overhead of broadcasting this value to all the processors is leading the shell-sort to have a worst performance than the Odd-Even sorting algorithm. I was not able to simulate the sorting for 2^{30} elements because the job only has allocation of 1Gb of memory on the grid. In order to guarantee that all algorithms are sorting the exactly same array, I needed to allocate memory in the beginning of the program and assign the value to all of them at the same time. That is where the memory crashed when there are 2^{30} elements.

In conclusion, with the results obtained it is possible to conclude that the best sorting algorithm for arrays that have 2^{20} or more elements is the Odd-Even sort in parallel.

Figure 6. contains a screenshot to prove that all the 3 implementation sort the random array correctly.

```
9 67 17 121 65 16 83 121 114 102 52 4 29 122 3 18 113 65 44 70 43 82 86 87 45 93 83 99 127 67 94 7
Qsort: 0.000004
 3 4 7 9 16 17 18 29 43 44 45 52 65 65 67 67 70 82 83 83 86 87 93 94 99 102 113 114 121 121 122 127
Shell Sort time:0.090982
3 4 7 9 16 17 18 29 43 44 45 52 65 65 67 67 70 82 83 83 86 87 93 94 99 102 113 114 121 121 122 127
Odd-Even sort: 0.031975
3 4 7 9 16 17 18 29 43 44 45 52 65 65 67 67 70 82 83 83 86 87 93 94 99 102 113 114 121 121 122 127
~
```

Fig. 6. Sorted elements by each algorithm