Performance Analysis of different Parallel sorting algorithms

Md Masrul Huda[1, 2], Edward A. Luke[2,3,*]

[1]Dave C. Swalm School of Chemical Engineering, Mississippi State University, MS State, MS
[2]Center for Advanced Vehicular Systems, Mississippi State University, MS State, MS
[3]Computer Science and Engineering, Mississippi State University, MS State, MS
[*] Corresponding author: luke@cse.msstate.edu

Sorting is a widely used technique in lots of computational works. Often time, a problem requires hundreds of thousands input is to be sorted in a convincingly faster time such as internet searching, data query in larger data station, clustering of data, artificial intelligence problems etc. There are a lots of choice available for sorting input entries such as heap sort, bubble sort, merge sort, insertion sort, bitonic sort, selection sort, sample sort or hybrid sort (mixed of multiple sorting techniques). These algorithms are initially developed from view point of serial execution. Even not all available sorting algorithms are parallelizable or they are not cost optimal or their iso-efficiency function is very poor. For these challenges, sorting is in great interest for HPC community. In this article, we present a comparative study of different parallel sorting implantation such as Parallel bitonic sort, Sample sort, hybrid Sample-bitonic sort and Quick sort inspired by hybercube architecture.

1.1 Parallel bitonic Sorting:

Bitonic sequence is a sequence where input data are monotonically increasing and followed by decreasing. Two sub-sequence are recursively compare split until sub sequence contains only one element. Due to its inherent nature is well suited for hypercube network. In this work, bitonic sorted is implemented for hyper-cube network. A pseudocode of the Parallel bitionic sort is given below from text book.

```
procedure Bitonic_Sort(label,d){
begin
    for(int i=0 to d-1){
        for(int j=i down 0){
            if((i+1)bit!=(j)bit){
                comp_exchange_max(j);
            }else{
                comp_exchange_min(j);
            }
        }
    }
}
```

1.2 Cost analysis(Hyper-cube Network):

Total Step= (1+log n)log n

Cost of each step=θ(1)
Parallel execution time= $\theta(\log^2 n)$
Parallel Cost=$\theta(n\log^2 n)$
Serial Counterpart cost=$\theta(n^2 \log^2 n)$
This algorithms is cost optimal with respect to it's seral implementation but not with respect to best serial cost is θ(n log n).

1.3 Cost for block of input data:
Each processor is assigned to n/p data, then data is sorted locally and followed by compare-split. Local sorting requires θ[(n/p) log (n/p)] and compare split requires $\theta(\log^2 p)$ steps.

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(\frac{n}{p}\log^2 p\right)}^{\text{comparisons}} + \overbrace{\Theta\left(\frac{n}{p}\log^2 p\right)}^{\text{communication}}$$

2.1 Sample-Sort:

Initially data is distributed over processes. Then data is sorted locally, p-1 evenly spaced sample is collected from each process, then combined p(p-1) sample is sorted each of the processes. Again (p-1) evenly spaced splitters are selected from p(p-1) sample. Then, Every process obtains the splitters, splits data accordingly and send them to corresponding bucket/process then data is sorted again locally. A pseudocode of the Parallel bitionic sort is given below from text book[1].

```
1  procedure Sample_Sort(local_list,n){
2  begin
3      sort(local_list)
4      find_splitters(local_list,local_splitters)
5      All_Gather(local_splitter,global_sample)
6      sort(Global_Sample)
7      find_splitters(Global_Sample,Global_Splitters)
8      Split_list(local_list,Global_Splitters,Splitted_list)
9      AlltoAllpersonalized(Splitted_list,concatinated_list)
10     sort(concatinated_list);
11 }
```

2.2 Cost Analysis:

Local sorting cost= θ[(n/p) log (n/p)]
Selecting local even spaced sample= θ(p)
Communication cost= $\theta(p^2)$
Internal sorting of p(p-1) data=$\theta(p^2 \log p)$

Selecting even spaced splitters= θ(p)
Bucketing the data=θ(p log (n/p))
Bucket transferring= 0(n) +0(p log p)
Final reorganizing the data= O(n/p)

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{local\ sort} + \overbrace{\Theta\left(p^2\log p\right)}^{sort\ sample} + \overbrace{\Theta\left(p\log\frac{n}{p}\right)}^{block\ partition} + \overbrace{\Theta\left(\frac{n}{p}\right)}^{communication}$$

From above equation, clearly we see p(p-1) sample sorting is the dominating term. Which results the iso-efficiency function =θ(p^3 log p).

To improve the iso-efficiency function a better approach can be adopted, p(p-1) sample can be sorted θ(p log p) time if we would use bitionic dort. Which has better iso-efficiency function θ(p^2 log p) .

3.1 QuickSort (influenced by hypercube architecture):

QuickSort often time suffers performance degradation due to poor pivot selection. A better approach is using some sampling technique. Here we describe a technique, where pivot is selected from median of local medians. This algorithm is implemented on hyper cube architecture, where pivot is used to exchange the data between two sub-cube. These sub-cubes are also recursively keep divided until there is only one process left. A pseudocode(Main part of code) code is described below.

```
double *parallel_quick_sort(double *buffer, int &loc_buf_size) {
  int d = log2(numprocs) ;
  sort(result_buffer, result_buffer+result_size) ;
  for(int i=0;i<d;++i){
    //Determine colors
    int color=myid/pow2(d-i);

    // Sort the local list followed by determining median
    my_median=result_buffer[result_size/2];

    //Split the communicator
    MPI_Comm_split(comm,color, myid, &new_comm);
    MPI_Comm_size(new_comm,&new_comm_size) ;
    MPI_Comm_rank(new_comm,&my_new_id) ;
    new_d=log2(new_comm_size);

    //Gather all median onto all rank
    MPI_Allgather(&my_median,1,MPI_DOUBLE,median_buffer,1,MPI_DOUBLE,new_comm) ;

    // Sort and determine the pivot
    sort(median_buffer, median_buffer+new_comm_size) ;
    pivot=median_buffer[new_comm_size/2];
    my_partner=my_new_id^(pow2(new_d-1));
    if(my_new_id<my_partner){
        send(UpperHalf of mine);
        recv(LowerHalf of partner);
        merge();
    }else{
        send(LowerHalf of mine);
        recv(UpperHalf of partner);
        merge();
    }
    MPI_Comm_free(&new_comm);
  }
}
```

3.2 Cost analysis:

Assume a good pivot choice would result n/p data exchange. The algorithm will proceed for log p steps. Each step involve (ts+tw)log (2^k) time for pivot gathering,where k is number of processor in current sub-cube. After that ts+n/p(tw+tc) time for data exchange and merge (Though in this work, a naïve and horribly inefficient concatenate technique is used). At last step (n/p)log(n/p) to sort local data. So parallel execution time as following:

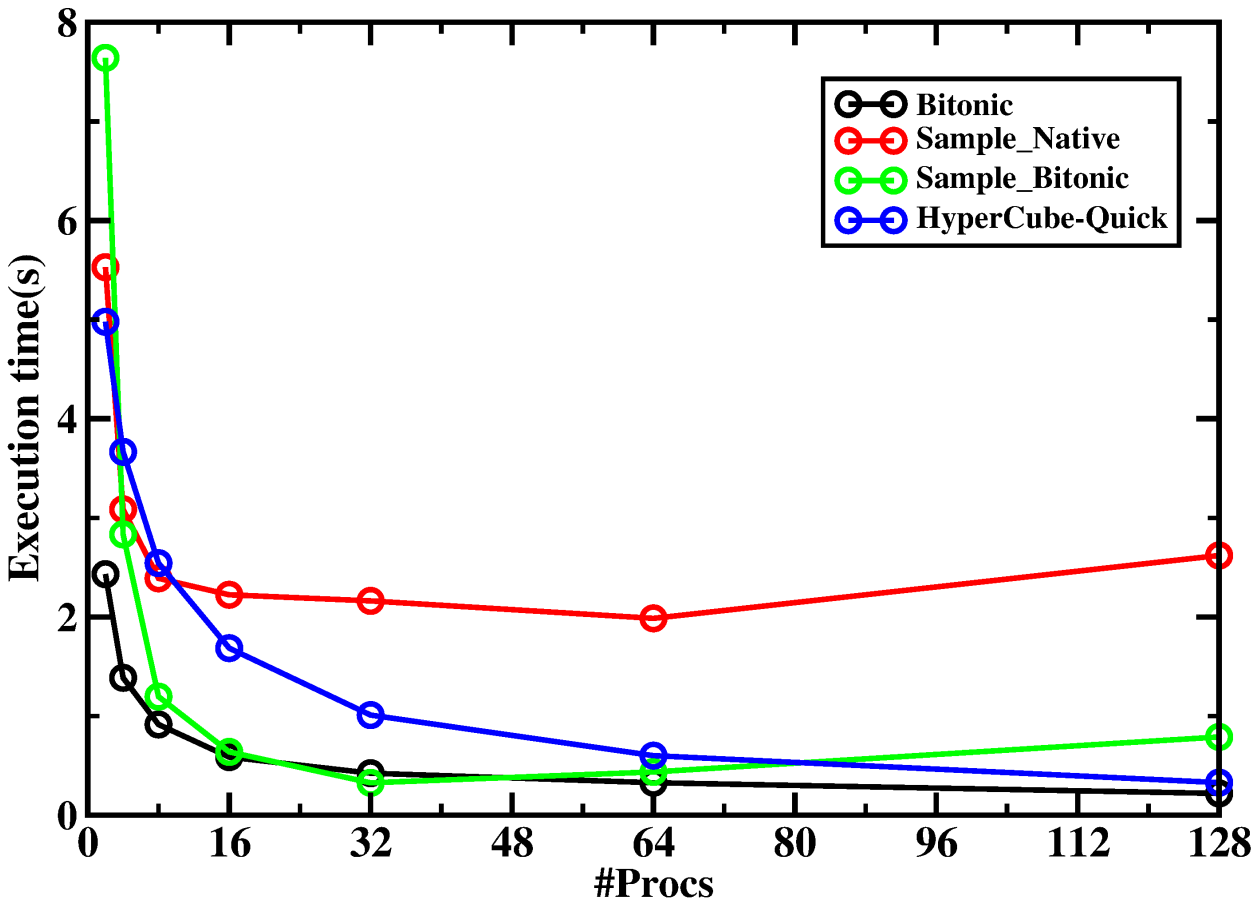
$$T_p = \sum_{k=1}^{logP} \left[(t_s + t_w) \log(2^k) + t_s \frac{n}{p} (t_w + t_c) \right] + t_c \frac{n}{p} \log \left(\frac{n}{p}\right)$$

ts= communication startup time
tw=Bandwidth term
tc= Commutation time

Simplified, asymptotic parallel execution time,
$T_p = \theta(\log^2 p) + \theta\left(\frac{n}{p} logP\right) + \theta(\frac{n}{p} \log \frac{n}{p})$

So, Iso-efficiency function of this algorithms is $\theta(p\log^2 p)$.

4 Result:



Performance analysis has been measure on shadow using intel-mpi. Input size was 50000000. Parallel performance of pure bitonic sort is very good whereas Simple sample sort performs very poor. But when Bitonic sort used, performance of the algorithms improved quite significantly. It

is obvious as main computational job is to sort p(p-1) splitters. In which case p(p-1) sample can be sorted $\theta(p \log p)$ using Bitonic sort compare to internal sorting of $\theta(p^2 \log p)$. Hypercube quick sort performs poor for lower processor count but it shows significant performance boost for higher processor count. As it has better iso-efficiency function among all, that's why these behavior is expected.