Parallelized Quicksort and Combinatorial Algorithms

* A CMSE822 Course Project

Chang Liu
Computer Science and Engineering
Michigan State University
East Lansing, MI
liucha39@msu.edu

***Abstract*—This paper will discuss 2 versions of paralleled Quicksort algorithm which implemented by MPI and OMP respectively. And there are 2 different strategies in the code including binary tree merges and better partition with prefix sum.**

***Index Terms*—Parallelization, Sorting, Prefix Sum, Binary Tree**

I. Introduction

There are many sorting algorithms but a few of them can be paralleled. For example, the classical bubble sorting compare and swap neighbour elements $n-1$ times which can not be paralleled because each iteration relies on the precious one. On the contrary, the Radix sorting can be paralleled because it computes the counting sort on each digit (of the maximum values in the array).

Quick Sorting is a classical algorithm which can be implemented by dynamic programming and recursion. The detailed algorithm will be presented in Methods part in this paper, but we can discuss its properties. Theoretically, the sequential quicksorting takes $n * log_2(n)$ time to execute which is faster than the bubble sort algorithm. However, due to the limitation of the maximum stack depth of the system, this algorithm can not deal with certain arrays of very large length.

In this paper, I will analyse the traditional Quicksort algorithm, along with MPI binary tree algorithm and prefix-sum based OMP algorithm. Both of MPI and OMP algorithms can achieve less complexity with the sequential algorithm.

The overall run time for sequential Quicksort to sort an array with about $100,000$ length is about 0.01s. By pluralization, this run time can be reduced. And there will be more detailed analysis in complexity part.

II. Methods

The methodologies here includes the origin sequential Quicksort algorithm, the MPI binary tree Quicksort and the OMP better partition Quicksort.

```
Sort(A)
    Quicksort(A,1,n)

Quicksort(A, low, high)
    if (low < high)
        pivot-location = Partition(A,low,high)
        Quicksort(A,low, pivot-location - 1)
        Quicksort(A, pivot-location+1, high)

Partition(A,low,high)
    pivot = A[low]
    leftwall = low
    for i = low+1 to high
        if (A[i] < pivot) then
            leftwall = leftwall+1
            swap(A[i],A[leftwall])
    swap(A[low],A[leftwall])
```

Fig. 1: Sequential Quicksort pseudocode

A. Quicksort Algorithm

The figure 1 shows the basic logics of the sequential Quicksort algorithm. First, we define a partition function (Figure 2)which chooses a pivot value from the current array and then move all elements less than the pivot to left side of the array; move all elements larger than the pivot the right side of the array. Next we partition the subarrays recursively of two sides. Finally, we sort the original array by $O(NlogN)$ (partition takes $O(N)$ times and each time the algorithm recursively quicksorts the array takes $O(logN)$ times.

1) The partition function: The boundary conditions of the partition function is quite important, in other words, the worst case that we may encounter during the Quicksort algorithm will pose effects on the overall performance on both sequential and parallel algorithms.

One possible case is that our pivot is exactly the smallest or the largest value of the current subarray. In this case, the parallel algorithm will degrade to the sequential algorithm because one side of the partition result is empty at all.

2) The Quicksort outer function: From the pseudocode, we can see that the algorithm utilizes a recursive structure to sort the subarrays of the original array. This function can be turned into a non-recursive function with a queue which push back

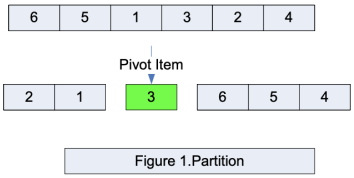


Fig. 2: A partition diagram by Qin [1]

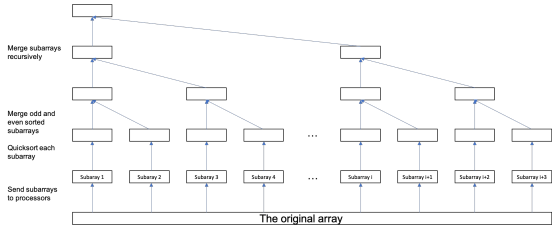


Fig. 3: The MPI based binary tree Quicksort structure

the left and right subarrays into the queue each time after the partition.

B. MPI distributed-memory based quicksort

The MPI based binary tree Quicksort simply segments the original array into processors and each processor sorts the subarray by the sequential Quicksort algorithm. Once all subarrays are sorted, we merge all subarrays by a binary tree structure, which is shown in Figure 3.

In the implementation, when merging 2 subarrays, the algorithm defines two pointers a and b, both starting at 0. By comparing the values of two subarrays, the algorithm put the $min(*a, *b)$ into the merged array and $argmin(*a, *b)\ += 1$.

If we assume the time complexity of the sequential Quicksort if $O(Nlog(N))$ and there are p total processors, then the total time complexity of the parallel algorithm will be:

$$\begin{aligned} n &= N/p \\ T &= \tau + N/p * log_2(N/p) + log_2(p) * N/p * (2 + 2^2 + \cdots + p) \\ &= \tau + N/p * log_2(N/p) + log_2(p) * N/p * (2 * 2^{log_2(p)} - 2) \\ &\approx \tau + N/p * log_2(N/p) + log_2(p) * N/p * 2p \\ &= \frac{1}{p} N * log_2(N) + (2log_2(p) - \frac{log_2(p)}{p})N + \tau \end{aligned} \tag{1}$$

From the time complexity above (Eq 1), we can deduce the boundary case of the algorithm

$$T = N/p * log_2(N/p) \quad \texttt{if} \quad N \gg p \tag{2}$$

In other words, if $N \gg p$, the parallel algorithm degrades to a sequential Quicksort (Eq 3)because there are not enough processors to do the sorting simultaneously.

$$T = 2N * log_2 N \quad \texttt{if} \quad N = p \tag{3}$$

Another extreme condition is that we get the same amount processors as the length of the original array, $N = p$. In this case, the total runtime takes about 2 times of the sequential Quicksort. One possible reason is that the merge part is less effective than the quicksorting each subarray.

We can also deduce the ideal best number of processors to apply for this parallel Quicksort.

$$\begin{aligned} \texttt{Given} \quad f(T) &= \frac{1}{p} log_2(N) + (2 - 1/p) log_2(p) \\ \frac{\partial f}{\partial p} &= -\frac{1}{p^2} log_2 N + \frac{2}{ln(2)p} + \frac{ln(p) - 1}{ln(2)p^2} \\ &= -log_2 N + \frac{2}{ln(2)} p + \frac{ln(p) - 1}{ln(2)} = 0 \end{aligned}$$

$$\begin{aligned} T \ \texttt{will the maximum when} \\ 2p + ln(p) = ln(N) + 1 \end{aligned}$$

$$\begin{aligned} \texttt{When } N \texttt{ and } p \texttt{ are both very large} \\ p \approx \frac{ln(N) - 1}{2} \end{aligned} \tag{4}$$

For the experiment, the dataset uses an array of 100000 elements. According to the above equation, we can calculate that the Parallel Quicksort will be most efficient when $p \approx 5.4$. Ideally, we don't need too many processors in order to get better performance on the parallel Quicksort. However, this equation does not take the communication time into consideration, so it's not easy to achieve the ideal speedup as we expect.

C. OMP shared-memory based quicksort

While MPI parallel Quicksort uses a binary tree to merge subarrays, the speedup can not achieve N/p if there are p processors. The equation 1 demonstrates that the extra time span is $\delta T = (2log_2(p) - \frac{log_2(p)}{p})N + \tau$ which will increase dramatically, especially when p is very large. So let's focus on another strategy that optimize the partition function which can reduce the time complexity from $O(N)$ to $O(logN)$.

Obviously, when partition a subarray, we should go through the whole subarray and swap those elements in wrong places. This will takes $O(N)$ time complexity. So, here the OMP Quicksort utilizes a parallel prefix sum algorithm to parallelize the partition function.

1) Parallel Partition function: In this part, we will focus more on how to parallel the partition part of the code by a good prefix sum function.

As Figure 4 shown, we first define an array *lt* which sets all index i with $A[i] < pivot$ to be 1 and $A[i] \geq pivot$ to be

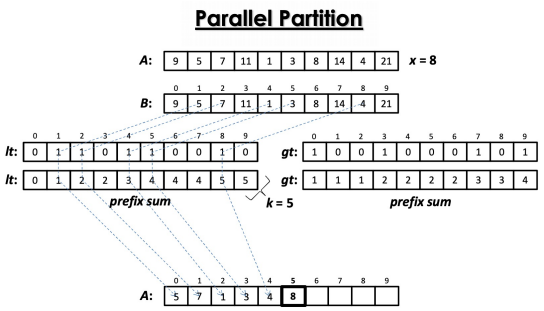


Fig. 4: The Parallel Partition function
Figure source: [2]

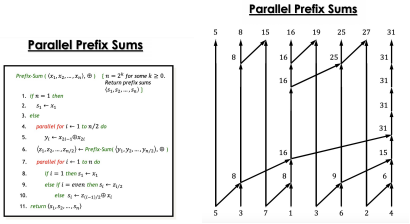


Fig. 5: The parallel prefix sum algorithm
pseudocode figure source: [2]

0. Next we calculate the prefix sum of the array *lt2*. Note the last element in *lt2* ($k = 5$) represents the correct position the pivot should be. Finally, all elements in prefix sum array *lt2* that are 1 in *lt* should be put into the partitioned array *A*. We can do it similarly for the right side of the prefix sum array *gt*.

2) The Parallel Prefix Sum algorithm: The following figure 5 shows that the algorithm uses a similar binary tree structure to calculate the prefix sum array. Note The Parallel prefix sum code is implemented referred to a github repository [3].

The time complexity of the sequential partition function is $O(N)$ because we should go through the whole array and swap elements in the wrong positions at least one time. But for the parallel Prefix Sum function, this can be done in $O(logN)$ times. Note that if we don't have enough processors (Assume p processors that less than the length of the subarrays n), the parallel prefix sum will only parallel execute on the first p elements and the rest of the elements will be prefix summed by sequential algorithm.

Assume the subarray length to be n, if $n > p$, then the run time should be $log_2(p) + n - p$, otherwise $log_2(n)$.

Ideally, if all partition operations can be done int parallel, the total run time of the Parallel Quicksort will be the equation 5, which can be the best performance for all parallel Quicksort algorithm, although it's not easy to achieve because we don't have as many processors as the input array becoming very long.

$$T = (log_2(N))^2 \tag{5}$$

```
-bash-4.2$ h5c++ quicksort_origin.cpp -o quicksort_origin.out
-bash-4.2$ ./quicksort_origin.out sequential.txt
The dataspace length is:100000
Read H5 successfully
Correct order with 0 errors
Run time of sequential algorithm is  0.010812
```

(a) Sequential

```
-bash-4.2$ h5c++ MPI_QuickSort.cpp -o MPI_QuickSort.out
-bash-4.2$ ./MPI_QuickSort.out mpi.txt
read h5 successfully
Correct order with 0 errors
Quicksort 100000 ints on 1 procs: 0.011954 secs
```

(b) MPI

```
-bash-4.2$ export OMP_NUM_THREADS=1
-bash-4.2$ h5c++ OMP_QuickSort.cpp -o OMP_QuickSort.out -fopenmp
-bash-4.2$ ./OMP_QuickSort.out omp.txt
The dataspace length is:100000
Read H5 successfully
Correct order with 0 errors
Run time of OMP parallel algorithm is  0.046748
```

(c) OMP

Fig. 6: The verification plots

III. Experiments

A. Project Structure

There are 3 main algorithms in the root directory including: *quicksort_origin.cpp*, *MPI_QuickSort.cpp* and *OMP_QuickSort.cpp*, along with a verification header file named *verify.h*.

In *data* folder, there are 2 HDF5 cpp files which can read and write HDF5 files respectively. Meanwhile, there are another MPI Parallel Quicksort which uses another structure but always fail to debug. And Finally, all results are stored directly in the *result* folder including plots and tables.

B. Data preparation

All data arrays to be used are created by create.cpp in data folder written by HDF5 library. In the experiment, the main algorithm reads an array consisting of $100,000$ random integers from 0 to 1000.

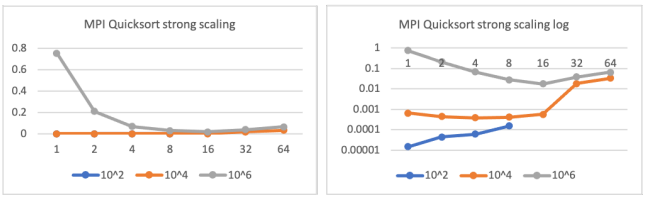
There is a file called multi.h5 which contains a dataset named *test* including the array generated by HDF5. And the main algorithms (including sequential, MPI, OMP algorithms) will utilize *dataset.read* to load the array.

C. Verification tests

At the end of each main algorithms, the verification header file is used to check if the sorted arrays are correct. And if it returns wrong orders, we can see how many errors there are in the sorted arrays. Also, to recheck the sorted arrays, the algorithms save their output into some *.txt* files which can be found in the root directory.

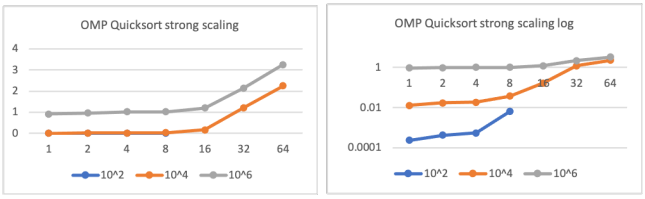
To verify whether the sorted arrays are correct or not, the algorithm includes the *verify.h* header file, and count how many errors in those outputs.

From the Figure 6, we can see there's no error at all in all 3 main algorithms. We can check the outputs manually by searching through the output files in *result/verify* directory as well.



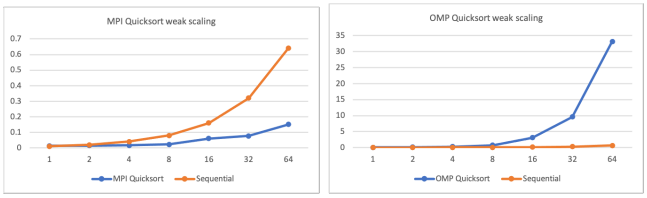
(a) The plot (b) The log plot

Fig. 7: MPI strong scaling



(a) The plot (b) The log plot

Fig. 8: OMP strong scaling



(a) MPI weak scaling (b) OMP weak scaling

Fig. 9: weak scaling plots with 10^5 vector length per processor

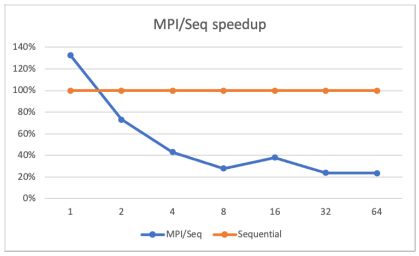


Fig. 10: The MPI/Seq speedup

D. Scaling performance

As Figure 6 shown, it takes about 0.01s in a sequential Quicksort algorithm which can be considered as a baseline for the parallel Quicksort algorithms.

1) strong performance: From the table I and figure 7, 8, we can conclude that The MPI algorithm can get a obvious speedup when the vector length increases. However, the OMP algorithm cannot get any speedup. The most possible reason is that the parallel prefix sum algorithm is not efficient even than the sequential prefix sum algorithm.

Next, it's not good to use large numbers of processors which may be slower than some small numbers of processors. The provement can be found in the previous methods part in this paper. For vector length 10^6 in MPI plot, the best processor nums are 16 and it's 4 for 10^4 vector length. We don't see any speed up in 10^2 vector length. One possible reason is that the array is too short and the communication time counts most in this situation.

Finally, for the OMP Quicksort, the run time increases along with the increasing number of processors. There maybe 2 reasons here: one is that the parallel prefix algorithm is not efficient; the other is that the communication for processors fails.

2) weak scaling: Figure 9 tells us that both MPI and OMP algorithms takes much more time if we uses more processors while keep the data array in each processor the same.

The MPI algorithm can still do better than the sequential Quicksort when processors increase (Figure 10) and it seems to remain at about 22% which proves that the MPI algorithm is quite stable and robust. The OMP algorithm totally fails to do the speedup, so it's unnecessary to plot its percentile relationships with the sequential Quicksort.

E. Memory usage and IO operations

The input of all algorithms are integer arrays which means for a array of length 10^6, it takes about $4Mb$ memory to just restore the input data. We will analyse the memory usage for all 3 allgorithms. Here we will assume the situation that the vector length is 10^6 for all 3 algorithms.

1) Sequential Quicksort: The sequential algorithm uses a recursive structure but all swapping operations are done on the same memory area starting at the same pointer. So the total memory usage here is

$$M_{seq} = O(N) \approx 4Mb \tag{6}$$

2) MPI Binary tree Quicksort: The MPI algorithm needs to sort subarray on each processors, so it takes extra $O(N)$ memory space. And then, to merge the subarrays through binary tree, we will need $2 * O(N)$ more space because each merge step requires a new temporary array to store the merged subarrays. So the overall ideal memory usage is

$$M_{mpi} = 3 * O(N) \approx 12Mb \tag{7}$$

3) OMP Quicksort: The parallel partition doesn't require extra memory space, so the memory usage will be the same as the sequential Quicksort ideally.

$$M_{omp} = O(N) \approx 4Mb \tag{8}$$

Vector Length	10^2		10^4		10^6	
Processors	MPI Quicksort	OMP Quicksort	MPI Quicksort	OMP Quicksort	MPI Quicksort	OMP Quicksort
1	0.000015	0.00024	0.000651	0.012898	0.754826	0.923674
2	0.000044	0.000425	0.000451	0.017534	0.210867	0.96964
4	0.000062	0.000546	0.000389	0.018821	0.068128	1.020204
8	0.000153	0.006341	0.000415	0.037859	0.028485	1.024894
16	-	-	0.000568	0.166941	0.017803	1.20806
32	-	-	0.018639	1.214909	0.039132	2.148102
64	-	-	0.033499	2.247199	0.06647	3.25347

TABLE I: Strong Scaling with different vector lengths

	MPI Quicksort	OMP Quicksort
1	0.013278	0.053079
2	0.014606	0.111082
4	0.017207	0.270608
8	0.022196	0.761439
16	0.060687	3.068236
32	0.076338	9.56862
64	0.150424	33.1644

TABLE II: Weak Scaling with different vector lengths

IV. Conclusions and Future work

A. Conclusion

1) The MPI binary tree Quicksort is quite stable and robust compared to the sequential Quicksort. The OMP partition Quicksort can do better once it uses a better prefix sum algorithm.
2) Both parallel Quicksort don't require large numbers of processors to achieve the best performance which means we don't need to apply a huge servers in order to do the parallel Quicksort. The equation 4 matches our experiment very well, so we can estimate how many processors it required even before sorting.
3) Considering the boundary case, we can ensure the MPI Quicksort won't takes more than 2 times the sequential Quicksort needs. For OMP Quicksort, it only needs $O((logN)^2)$ times, but due to the bad performance of the prefix sum algorithm, we can't see the speedup in this experiment.

B. Future work

First, The OMP partition Quicksort requires a better parallel prefix sum algorithm in order to achieve good performance.

Also, in the MPI Quicksort algorithm, *MPI_Send* and *MPI_Recv* are used to communicate which can be modified with a non-blocking structure. And the OMP Quicksort uses the default schedule directives which can be replaced with dynamic parallel directives.

Finally, the partition function can be improved. Since we only choose a pivot and move those smaller to the left and larger to the right, it may not be very efficient when we have many idle processors. One possible improvement is that we choose multiple pivots and segments multi times which can utilize those idle processors in the binary tree structure.

References

[1] S. Qin, "Quick sort algorithm." Florida Institute of Technology Melbourne, FL 32901, 2011. [Online]. Available: https://cs.fit.edu/ pkc/classes/writing/hw15/song.pdf

[2] R. A. Chowdhury, "Parallel quicksort and selection." Department of Computer Science SUNY Stony Brook, 2013. [Online]. Available: https://www3.cs.stonybrook.edu/ rezaul/Spring-2013/CSE638/CSE638-lectures-8-9.pdf

[3] sagargoyal96, "Parallel-convex-hull-and-prefix-sum." [Online]. Available: https://github.com/sagargoyal96/Parallel-convex-hull-and-prefix-sum