# Programming Assignment 2: Parallel Quicksort

Qi Zheng (qzheng61); Wenqing Shen (wshen35)

April, 2017

## Abstract

In this assignment, we implemented parallel Quicksort with random pivoting. The first section of this report provides a brief description of the parallel algorithm and implementation procedure. The computation results using clusters and corresponding analyses are presented in the second half. At the end, discussions and ideas of performance optimization are proposed.

## Method

1. In a parallel quicksort problem, we have $n$ numbers to be sorted using $p$ processors. Each processor has $\left\lfloor \frac{n}{p} \right\rfloor$ or $\left\lceil \frac{n}{p} \right\rceil$ numbers locally at first.
2. Processor 0 (Rank=0) generates a random number $m$ in the range of $[0, p-1]$. Assume processor rank m has $nm_{local}$ local numbers. Then rank m generates a random number in the range of $[0,\ nm_{local}-1]$ as the index for the pivot. Broadcast the pivot to all processor in the current communicator.
3. We define a **array_partition** function to partition local numbers on each processor. Put number less than or equal to pivot at the left side, and put number larger than pivot at the right side. Use $n_{left}$ to count the number that is less than or equal to pivot, and $n_{eq}$ for the number that is equal to the pivot. There are $n_{right}$ numbers at the right side.
4. Get total count $n_{left-total}$ of number that is less than or equal to the pivot by MPI_Allreduce. Use the same way to get total counts $n_{eq-total}$ that is equal to the pivot. Total count of number larger than pivot is $n_{right-total}$. Partition the processors to two groups, $p_{left}$ and $p_{right}$, according to the ratio of $n_{left-total}$ and $n_{right-total}$. If $n_{eq-total}$ equals total number of the current problem, which means all numbers in the current sub-problem are equal, then we don't need to sort them, just return the length for further processing(recursion). If any size has more than zero numbers to be sort, but gets zero processor as a result of a small ratio, take one processor from the other side, and assign to this side to ensure no leave-out numbers in sorting.
5. Based on $n_{left}$, $n_{right}$, $p_{left}$ and $p_{right}$, we can decide how to distribute the local number to proper processor by **MPI_Alltoallv**. First get send_count array, use it to get receive_count array. These two count arrays could be used to get displacement array, thus all input info to MPI_Alltoallv has been assembled.
6. Split the current problem to two sub-problems, assign $p_{left}$ and $p_{right}$ number of processors to handle two subproblems.
7. Recursively run step 1~ 6 until all subproblems only has one processor to handle, then use sequential quicksort to sort local array.

8. Finally, distribute the sorted numbers evenly to all processors. This is our result for final output.

## Results and Discussion

Our codes could deal with different type of $n$ and $p$, even or odd, the power of 2 or not. It can also deal with the case with $n < p$. In this report we show the effect of $n$ and $p$. $n$ is from $1e2$ to $1e8$, $p$ is from 1 to 64.

### *Discussion about Runtime*

Figure 1 shows the run time with different processor number. When $n$ is small, the run time increase with $p$. When $n$ is large, with $p$ increasing, the run time decreases first and then increases. When n is small, the communication time is the main part. Larger p will need more problem split and takes more communication time. When n is large, p is small, though communication take a little time, the computing time is much less, thus the total time decrease with p. When n is large and p is also large, then the communication time takes the majority.

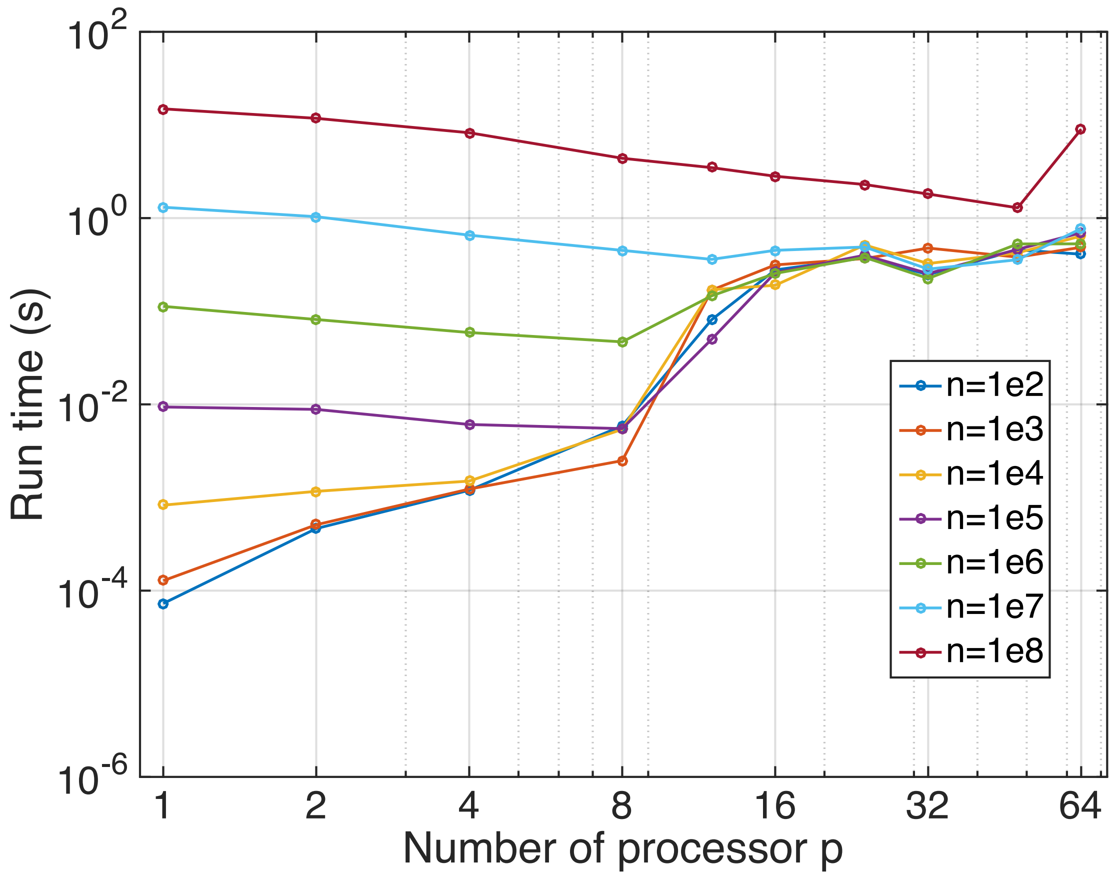

Figure 1. Run time with different processor number

Figure 2 shows the run time with different input length. The run time increases with n for certain p, because both communication and computing time would increase. For small p, the run time increase with n rapidly; while for large p, the run time doesn't change much for n<=1e7. The reason might be that, for large p, and not too large n, the latency for communication consumes the most of running time.

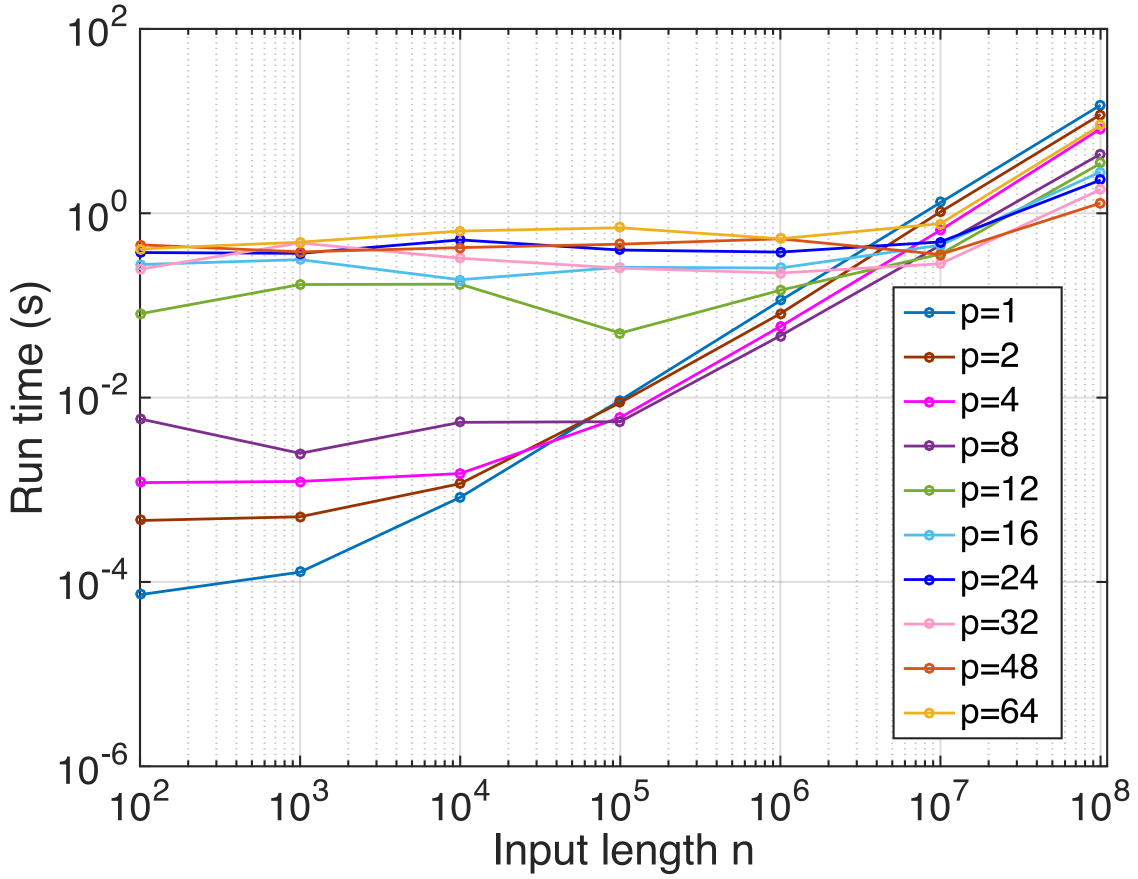

Figure 2. run time with different input length

## *Discussion about Speedup*

Figure 3 shows the speed up with different processor number. For the tested cases, to have speed up larger than 1, large $n \geq 1e5$ is needed. The benefit of saving computing time by multiple processor is only obvious when problem size is large, when the computing time saving would be greater than additional time consumption by extra communication. We note that for $n \geq 1e7$, speedup is always larger than 1, though according to the trend, the speedup might be less than 1 if using much more processor like $p = 100$. For $n = 1e5$ and $1e6$, the speedup is larger than 1 for limited processor number, because more processor might make the communication time the majority. While for $n < 1e5$, the speedup is always less than 1, because the computing time for any $p$ is relatively small compared with communication time. Comparing different n and the range of $p$ that could achieve large speedup, we found that the larger the n is, the wider range of p that could have large speedup.

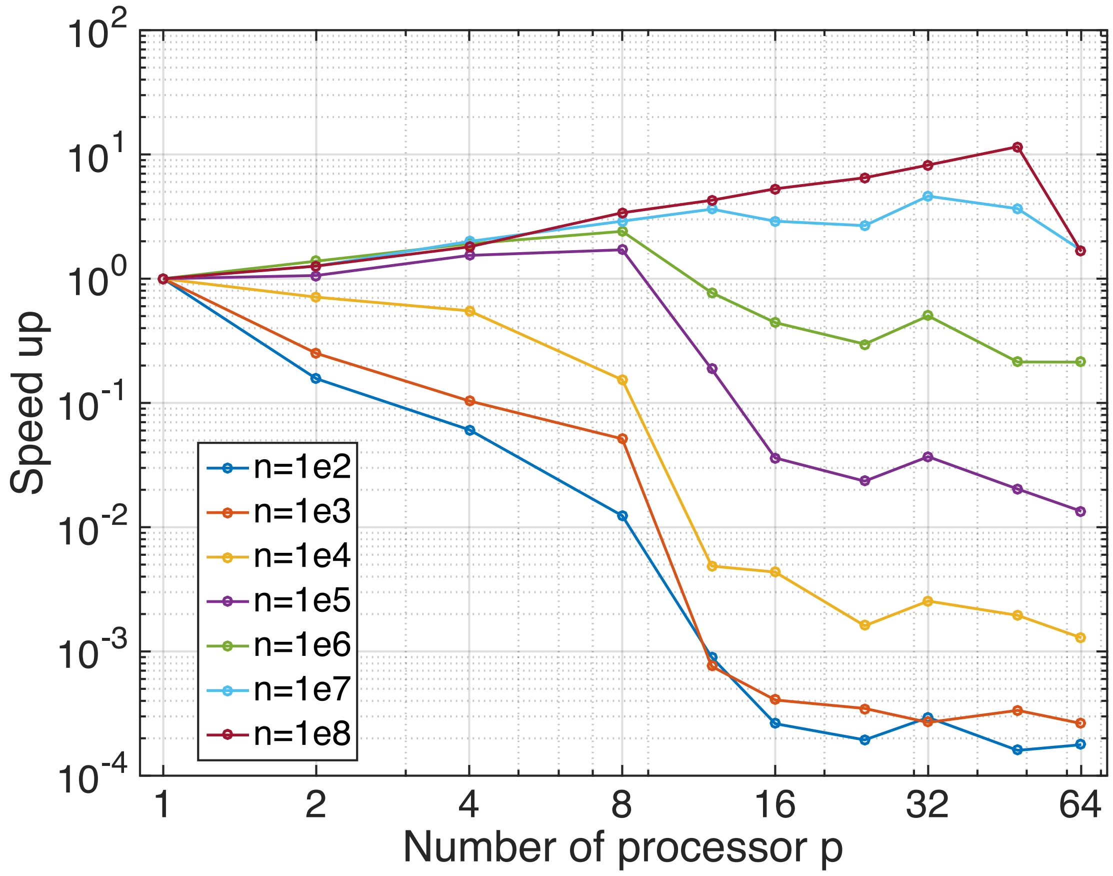

Figure 3. Speed up with different processor number

## *Discussion about Efficiency*

Figure 4 shows the efficiency with different processor number. The efficiency would decrease with $p$ for certain $n$. When there are more processors, the communication time increases due to increased problem split and increased time of communication (related with $log\ p$), while the total computing work are the same for certain $n$. For certain processor p, the efficiency decrease with n, still due to increased communication time, caused by more data passing time (related with $n/p$) for each communication. To get good efficiency, according to the test, $n \geq 1e5$ is required for $p \leq 8$, $n \geq 1e7$ is required for $8 < p \leq 48$.

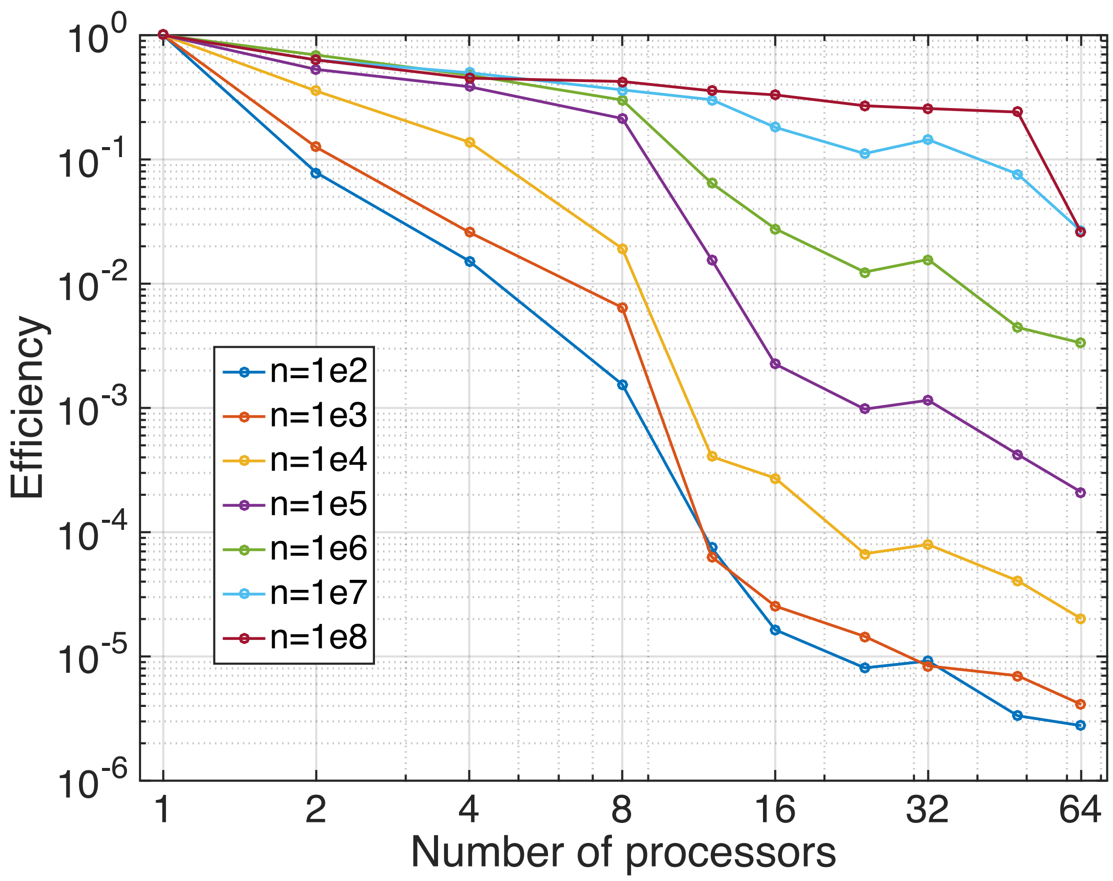

Figure 4. Efficiency with different processor number

To better understand when optimal efficiency could be achieved. Figure 5 shows effect of local size $n/p$ on efficiency. For certain $n$, the efficiency increases with n/p, which means large local size would benefit the efficiency. For larger n, larger minimum n/p is required to achieve good efficiency. For $n = 1e6$, minimum $n/p$ is around $1e5$; for $n = 1e8$, minimum $n/p$ is around $2e6$.

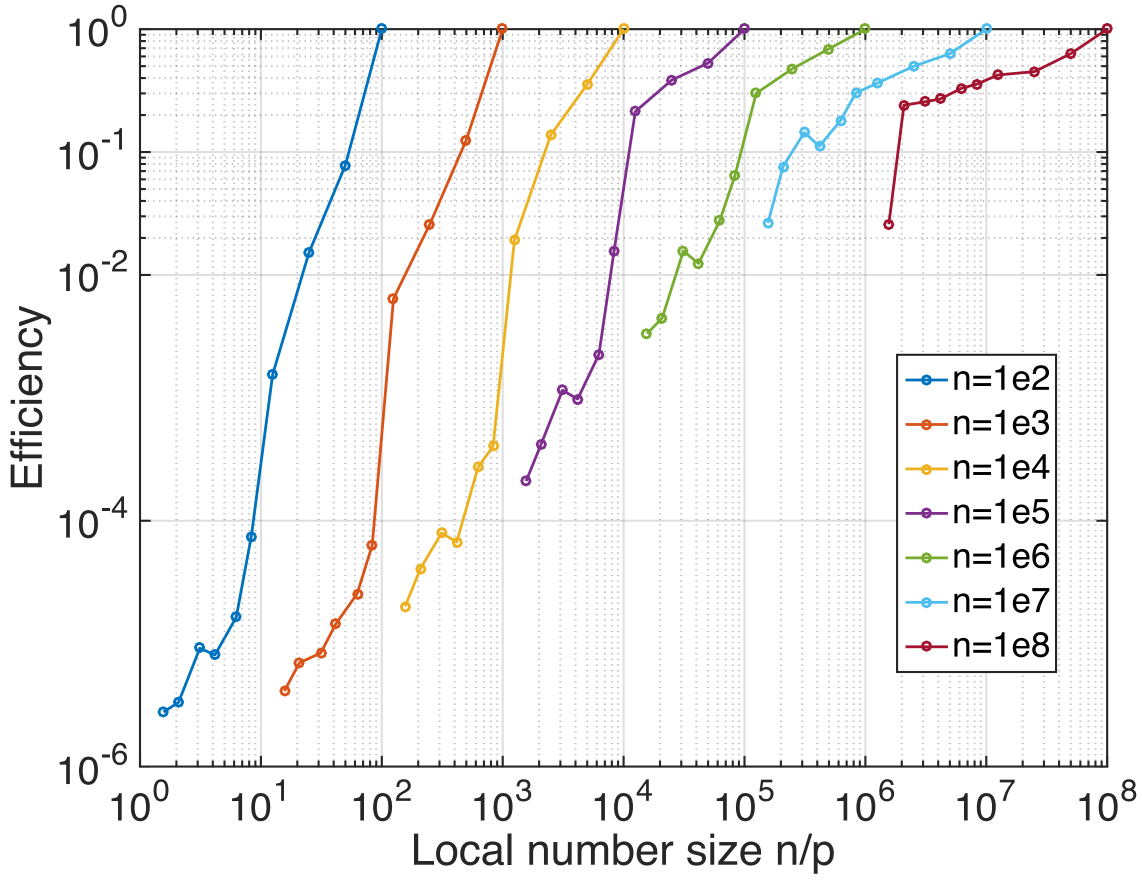

Figure 5. Effect of local size $n/p$ on efficiency for certain n

If fixing $p$, and studying the effect of $n/p$ on efficiency. The results are plotted in Figure 6. For certain p, large $n/p$ would get better efficiency. For $p = 3$, minimum $n/p$ is around $2e3$; for $p = 12$, minimum $n/p$ is around $2e5$. While for $p = 64$, however, all the cases have efficiency less than 0.1.

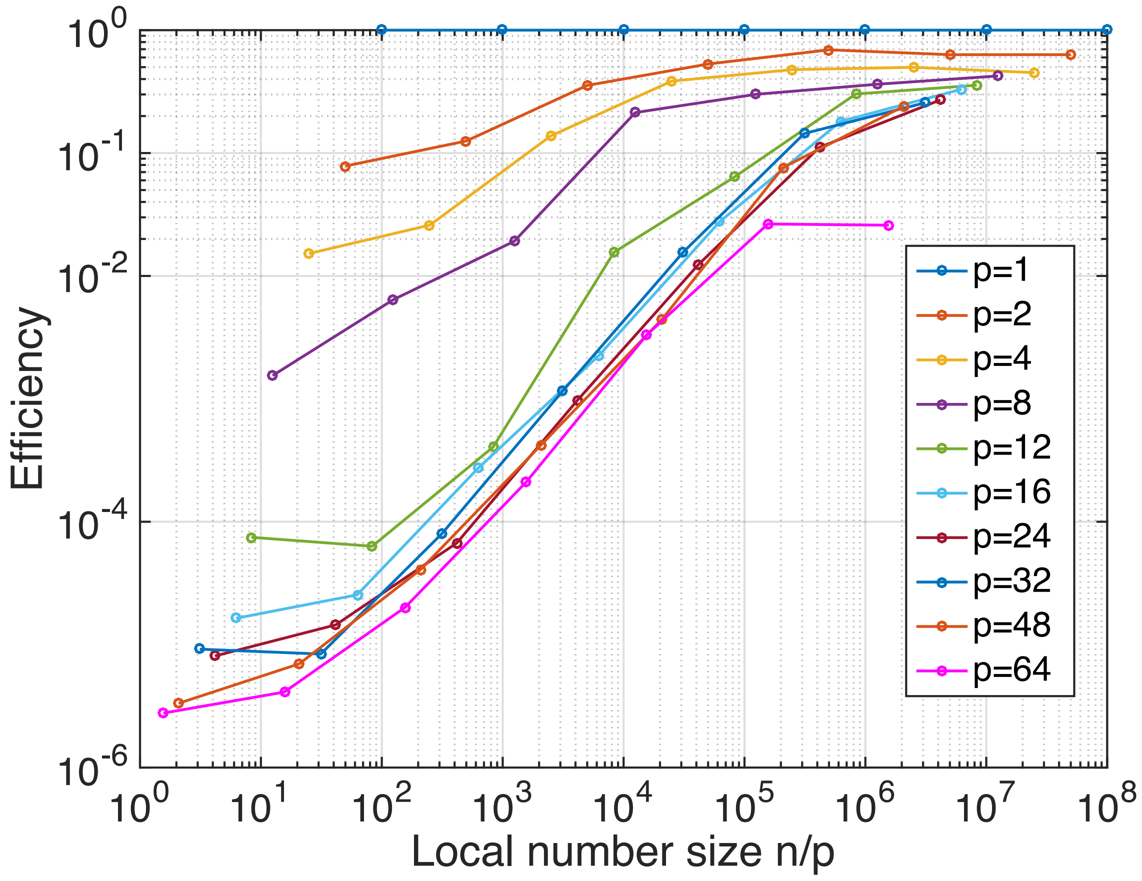

Figure 6. Effect of local size $n/p$ on efficiency for certain p

## Conclusion

Large $n$ is required to have large speedup. To achieve optimal efficiency, large local size $n/p$ is necessary. Larger n results in larger minimum $n/p$ to get optimal efficiency. Since in parallel implementation of Quicksort, the role of communication time is significant, better network could improve the performance of parallel sort. In the implementation of parallel sort, choosing proper pivot is important, thus better way to pick pivot would fasten the sort.