Project #3 of High Performance Computing

Xin Xing, Chengwei Li

Georgia Institute of Technology

I. Algorithm implementation

Basically, our implementation follows the instruction in the assignment and the program structure given. Additionally, in order to get better performance over efficiency and storage, several improvement is made in our program as introduced by the following.

Pivot choosing

There is a problem discussed in the Piazza that some student observed that the RNG(random number generator) with same seed may generate different sequences at different nodes.

Besides, the introduced method in the project also requires each processor have the same random seed when generating pivot. The general idea is to use some common integer shared by all the processors in the communicator such as the number of processor p or any randomly chosen constant(like 0). As a result, this scheme is virtually not totally random for different problem as the RNG sequence are fixed.

Hence, we generate the pivot by following scheme,

1. Set the random seed by current time by `srand(time(NULL))`.
2. In root processor, generate the processor rank m randomly by `rand()%numProc`.
3. Broadcast the generated m.
4. In rand-m processor, randomly choose one of its local elements as pivot by `rand()%localSize`.

In this way, we can always generate totally random pivot for any different problem even with same length n and same number of processors p. And it also avoid the possible error appears in different node with same seed.

Redesign of the C++ function parallel_sort

In the given program, the major sorting function is defined as

```
void parallel_sort(int* begin, int* end, MPI_Comm comm)
```

whose main disadvantage is that the array `[begin, end)` is fixed that we cannot resize or redirect the pointer. This is mainly intended to the in-place sorting. However, in the parallel Quick Sorting, we need to irregularly partition the array that the size of local array may change. In-place sorting is not quite feasible in our case.

A natural way to implement this function is to store the data in a temporary array `tmp` after partition, then recursively implement `parallel_sort}` to `tmp` in the partitioned communicator, at the end, copy the `tmp` back to `[begin, end)}` array.

One problem about this implementation is the storage usage piling up along the recursive call of `parallel_sort}`. The `tmp` cannot be deleted until the related sorting `parallel_sort` over it is finished. So, there will be a increase of storage use in each call of `parallel_sort()`. As we've known that, for a general array, we need to recursively call the `parallel_sort` for at least $O(\log p)$ times in some processor. Assuming that all the partition is uniform enough that size of `tmp` is always $O\left(\frac{n}{p}\right)$, the storage in some processor goes up to $O\left(\frac{n}{p}\log p\right)$. As a result, as p grows big, the extra storage needed for this implementation is non-negligible.

In order to change the size of local array inside the `parallel_sort}`, we redesign it as another function

```
int parallel_sort_changeable(int* &begin, int size, MPI_Comm comm)
```

In this function, we still use the temporary array `tmp` to store the reallocated data after partition. Then, we just need to resize the original array and copy `tmp` back to it. After this, we can delete the `tmp` array right before the recursive call `parallel_sort_changeable` over the original array `[begin, begin+size)`. In this way, the storage won't pile up along with the recursive call. Besides, this function returns the new local size of the pointer `begin`.

II. Analysis of Quick Sorting

Given a length-n array, denote $T(n,p)$ as the runtime of sorting it by p processors. Using the pivot, we can separate the array into two sub-arrays of length s and $n-s$ where s is decided by the pivot and the array. Meanwhile, in the partition and data reallocation procedure, $O\left(\frac{n}{p}\right)$ comparison is the main computational cost as `MPI_Alltoall` and `MPI_Allgather` dominate the communication runtime.

Only consider the computational cost, we know that

$$T(n,p) = \max(T(s,p_s), T(n-s,p_{n-s})) + O\left(\frac{n}{p}\right). \tag{1}$$

where p_s and p_{n-s} is assigned proportional to the array size s and $n-s$. Hence, the computational cost is mainly affected by partition times we need. Ideally, we only need $O(\log p)$ partition when every pivot separates the array uniformly. In the worst case, if the pivot only separates the array with 1 and $n-1$ in each partition, then we need $O(p)$ partitions.

To summarize, the expected computational cost is $O\left(\frac{n}{p}\log p + \frac{n}{p}\log\frac{n}{p}\right)$ and the worst case is $O\left(n + \frac{n}{p}\log\frac{n}{p}\right)$ where the $O\left(\frac{n}{p}\log\frac{n}{p}\right)$ is the locally sorting computation in exactly one processor. And the pivot choosing plays an important role to decide the computational cost.

III. Numerical Experiment

Basically, in our numerical experiment, we run all the tests by generating the local vector randomly with the command `mpirun -np <p> ./sort -r -n <n>`. In the following, we tested several aspects of the program.

Randomness

There are two positions in our program that introduce the randomness into our test. The first one is the randomly generated local sequence, while the other one is the randomly generated pivot. These two randomness can affect the computation time at each subproblem as well as the number of recursive call of the sorting function. Hence, in our test, when we simply test once for each group of configuration (n, p), we get the Figure 1 showing the strong disturbance between running time and array size n.

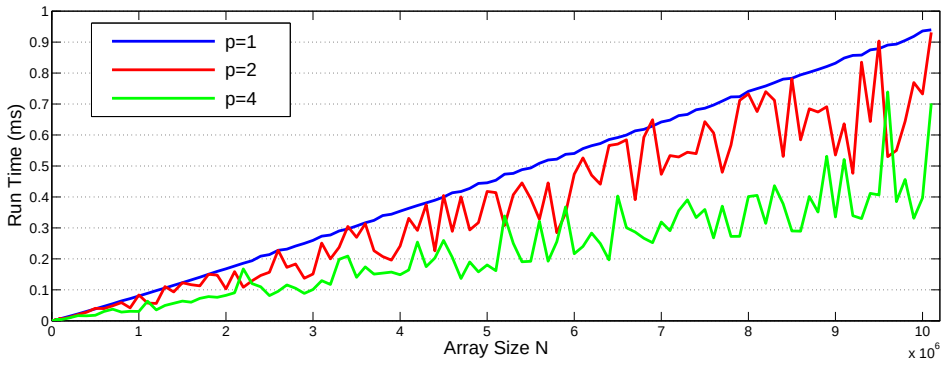

Figure 1: *Result without averaging*

To better show the property of the parallel quick sort, the average running time should be collected by repeatedly sorting the same array with random pivot. With this idea in mind, we repeat the sorting over each problem for 20 times and collect the running time. What we get is obviously clearer relationship between array size and runtime in Figure 2.

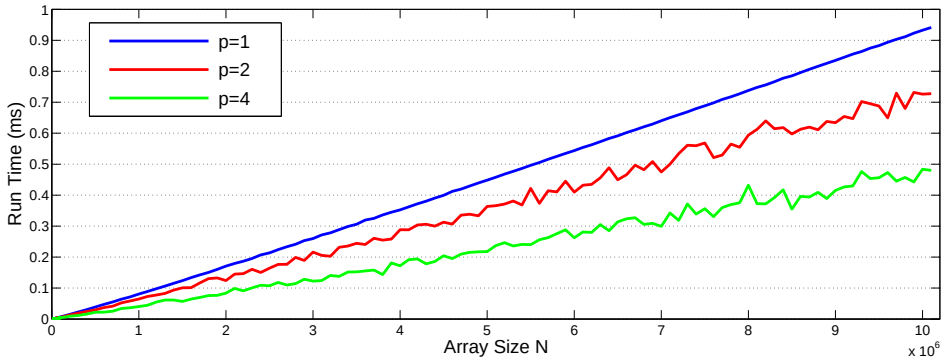

Figure 2: *Result with averaging*

Besides, all the results in the following subsections are obtained by averaging the result of 20 times repeat test.

Basic Result

Generally, we applied our program over $p \in \{1, 2, 4, 8, 16\}$ with array size $n \in [10^2, 10^7]$. The run time v.s. array size is drawn in Figure 3. There are several observations we can make from it as following.

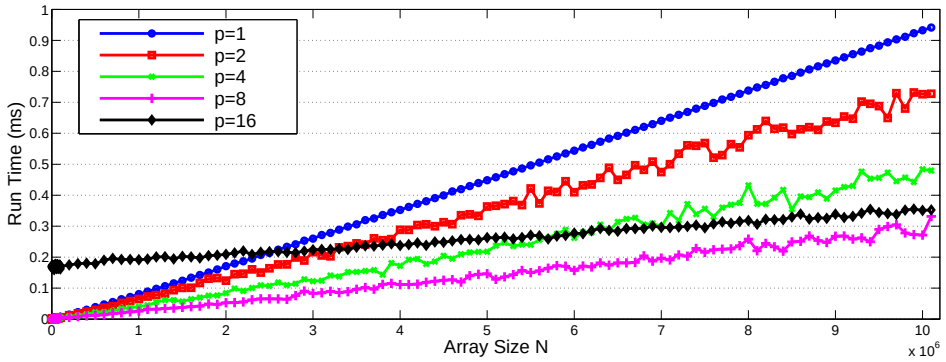

Figure 3: *Runtime v.s. Array size n*

- For $p = 1$, the curve is quite smooth as there's no randomness's effect in single processor. Besides, as in this case, the program is using `std::qsort()` which has $O(n \log n)$ complexity. So, through statistical regression, we may be able to verify it.
- For $p = 2, 4, 8$, the benefit of parallel computing emerges as less run time is used with the same-size array.
- For $p = 16$. The change of runtime along with the array size is much slower. When the array size is small (under 2×10^6), the runtime is much higher than those of fewer processors. As n grows larger, the benefit of faster computation arise and at the end ($n \approx 10^7$) it gradually outmatches $p = 2, 4, 8$. One highly possible reason is that the communication in the program dominates the run time.

  With fixed processor number p, the number of recursive partition is approximately $O(\log p)$ and bounded by $O(p)$. Hence, the communication among processors satisfies

  $$CommunicationTime \sim O\left(\tau C(p) \log p + \mu C(p) \frac{n}{p} \log p\right)$$

  where $C(p)$ denotes the number of communications in one call of the `parallel_sort` function. Besides, considering the possible congestion in the `MPI_Alltoallv`, the communication time may increase faster as p increases.

  To summarize, for bigger p like 16, the communication time begins to dominate the runtime while computation time is relatively small. However, the communication time is approximately linear to array size n while computation time is linearithmic to n ($n \log n$). Hence, when n grows bigger, the benefit of parallel computation will emerge just as shown in $n \approx 10^7$.
- There's an increasing disturbance as n grows. Variation of the runtime increases as n grows. More repeat times is needed for larger n to better estimate the expected performance of the program.

Speedup and Efficiency

Based on the basic result in the last subsection, the change of speedup and efficiency can be described in the Figure 4, 6, 5, 7, where Figure 4 and 5 shows the speedup and efficiency when

$n \in [10^3, 10^7]$ as the general trend and Figure 6 and Figure 7 shows them when $n \in [10^3, 10^5]$ for some detailed change when array size is small.

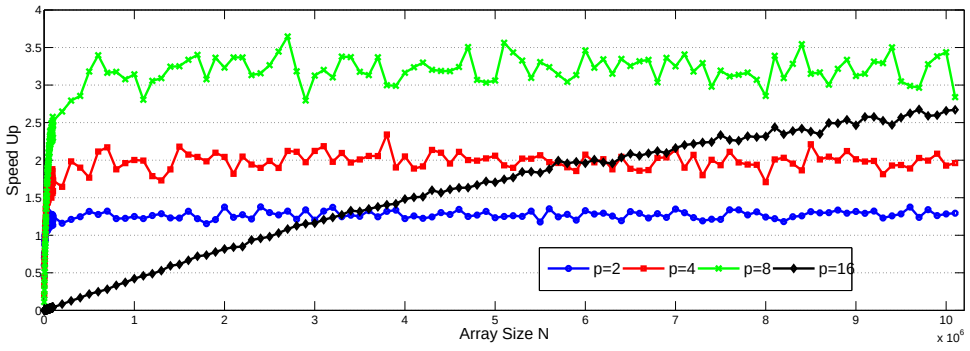

Figure 4: *Speed Up v.s. Array size n in* $[10^3, 10^7]$

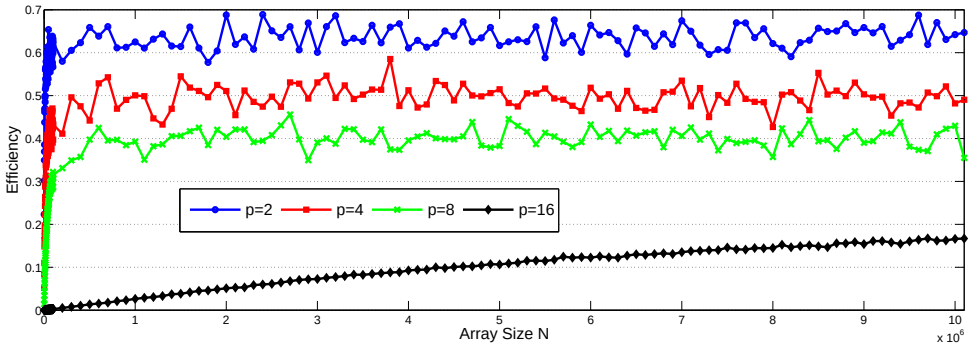

Figure 5: *Efficiency v.s. Array size n in* $[10^3, 10^7]$

Generally, we can get following conclusion about the speedup and efficiency.

- In all the test cases with $p = 2, 4, 8, 16$, the speedup and efficiency increase at first and then remain stable after n grows to some threshold.
- From all the information we know, the final stable speedup is larger when there're more processors. In the contrary, the final stable efficiency is smaller when there're more processors. The larger speedup is due to the faster computation based on parallel algorithm. The smaller efficiency is due to the more communication among the processors.
- When p is large, the array size threshold for reaching the stable or maximum speedup and efficiency is larger. And in fact, in our case, the result of $p = 16$ is still increasing and hasn't reached the maximum or stableness yet.

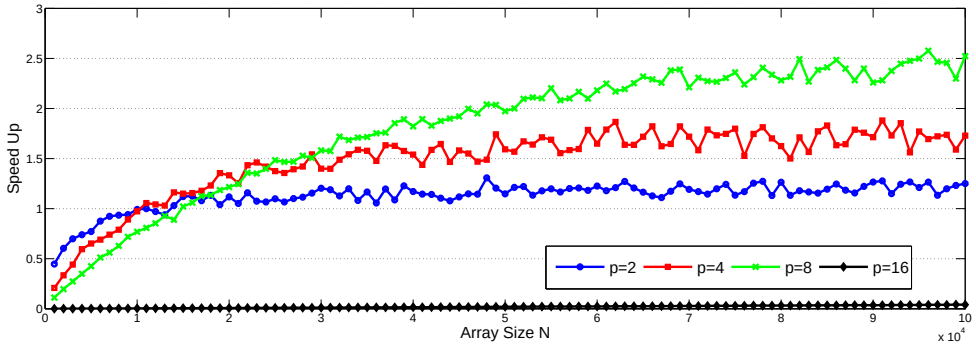

Figure 6: *Speed Up v.s. Array size n in* $[10^3, 10^5]$

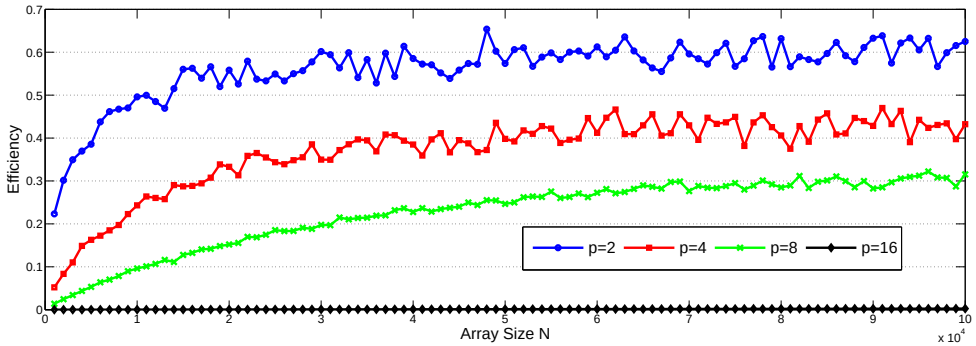

Figure 7: *Efficiency v.s. Array size n in* $[10^3, 10^5]$