UPPSALA
UNIVERSITET

Assignment 3

Parallel and distributed programming 5 credits 1TD070 62007

Group members:

Ibrohim Hamoud   ibrohimmn1994@gmail.com

Uppsala

March 19, 2022

1 Introduction and Problem description

This work deals with a parallel implementation of the quicksort's algorithm using MPI. The quicksort's algorithm revolves around choosing a pivot number of the array that can be chosen to be the median of the array, the mean, or even a random element from the array. The algorithm proceeds with partitioning the array around the pivot such that all the elements that are smaller than the pivot value are placed before the pivot and all the elements that are larger than the pivot go ahead of the pivot element. This procedure is the key process in the algorithm and can be summed up as putting the pivot value in its correct sorted order. The algorithm progress by applying the same procedure recursively on both sides of the partitioned array.

The parallel version of the quicksort's algorithm commences with dividing the array into equal parts and distributing them among the processors. Each processor applies locally the serial quicksort's algorithm on its share of the array according to a specific pivot strategy. Here, the pivot is chosen to be the global pivot across all array elements. The succeeding step is to divide all the processors into two groups and exchange the elements pairwise between the two groups, such that the processors from the first group receives the elements that are smaller than the global pivot while the other group receives the larger elements. At this point, each processor contains its remnant elements and the elements that are received. Thus, both sets of elements are then merged in a sorted way locally in each processor. The above-mentioned steps are repeated for every recursion step. An outline of the parallel quicksort's algorithm that is implemented in this work is depicted in Algorithm 1.

The pivot methods that are considered in this work are the following,
- Method 1: Select the median in one processor in each group of processors.
- Method 2: Select the median of all medians in each processor group.
- Method 3: Select the mean value of all medians in each processor group.

Algorithm 1

```
1) Divide the array into equal parts and distribute them with
    MPI_Scatterv()

2) Apply serial quicksort algorithm locally in each processor
   on its respective local array

3) start the parallel part of the algorithm:
    1] Compute the pivot
    For the mean method:
        - Choose the pivot to be the median on one local array.
        - Use MPI_Bcast to distribute the pivot to all other processors.
    For the median method:
        - Compute the local median.
        - Use MPI_Gather() to collect the medians into the
          root processor.
         - Sort the medians and compute the global median.
         - Set pivot = global median.
         - Use MPI_Bcast() to distribute the pivot.

    For the mean of the medians method:
        - Follow the same previous method except setting
          pivot = sum of the local median/ number of processors.

4) Locally, find the index of the first element that is larger
   than the pivot value. Thus the element of that index will
   serve as a local pivot in each processor.

5) Divide the processors into two halves. Halve the
   processors send the elements that are larger than the local
   pivot to the processors of the other halve and receive the
   elements that are smaller than the local pivot. The procedures
   is executed using MPI_Send/Recv pairwise between the the two
   halves depending on the rank. For 4 processors. The ranks 0
   and 2 pair together, while the rank 1 pairs with rank 3.

6) Locally in each processor,both the received elements and the kept
   elements which were not sent are stored in separate arrays. These
   two arrays are merged into a sorted order.

7) Divide the communicator into two separate sub-communicators
   using MPI_Comm_split() and apply the steps 3-7 recursively using.
```

2 Results and discussion

For an array that is ordered backward, the array in the first method is not divided into an equal part and thus the work-load balancing is poor for other pivot strategies. This can potentially explains the poor job of the first pivot method comparing the other two. Whereas the second and the third method take the median and the mean of the local medians, respectively, which shows an obvious better performance. However, for randomly ordered array, the opposite affect takes place due to a better work-load. (q.v 1 and 2).

Finding the median is more expensive on a parallel algorithm as the array of the medians needs to be sorted after being collected from all the processors. The remedy is to choose a pivot value that is close to the median. Thus, the third method plays the trick by computing the mean of the median which is considerably cheaper but at the cost of slightly worse load balancing. This can explain the slight difference between the second and third pivot methods in terms of performance.

The first pivot method(Figure 3) shows worse performance in comparison to other methods. This declining in performance indicates a worsening work-load balance for the first method as the size of the array increases. Here, the number of processors are scaled proportionally to the problem size. The plots show a linear increase in time execution with the size of the problem. Although the number of processors was scaled up proportionally. this is expected behavior as the parallel part scales linearly with the number of resources due to the communication cost that arises between the processors.

Table 1: Data for strong scaling given for a randomly sorted array of size 120×10^6.

Number of processors	Method1	Method2	Method3
4	8.737546	8.202274	8.208410
8	5.360150	5.387656	5.426944
16	2.169989	2.169510	2.163396
32	2.033136	2.322253	2.138083
64	3.146379	2.211514	3.425382

Table 2: Data for strong scaling given for a randomly sorted array of size 120×10^6.

Number of processors	Method1	Method2	Method3
4	4.636614	4.429344	4.703365
8	3.630761	3.463651	3.416813
16	1.583420	1.310084	1.305848
32	2.483412	1.856177	1.499929
64	3.858518	2.761287	2.301233

Table 3: Data for strong scaling. Here, The arrays used as an input to quicksort's algorithm are randomly sorted

Problem size	Processors	Method1	Method2	Method3
125×10^6	16	5.360150	5.387656	5.426944
250×10^6	32	3.832842	3.489427	3.542960
500×10^6	64	4.952221	6.822811	6.715610
1000×10^6	128	8.465559	9.795572	9.996629

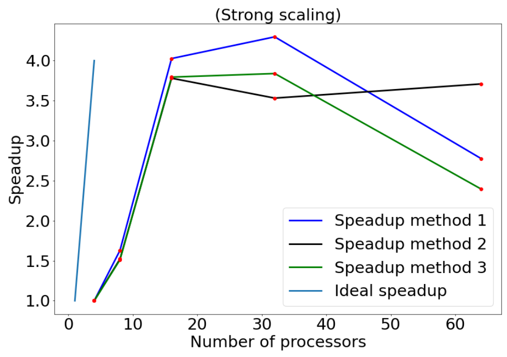

Figure 1: Plots of the speedup of different pivot methods as function of processors' number. The plots are taken for a randomly ordered array.

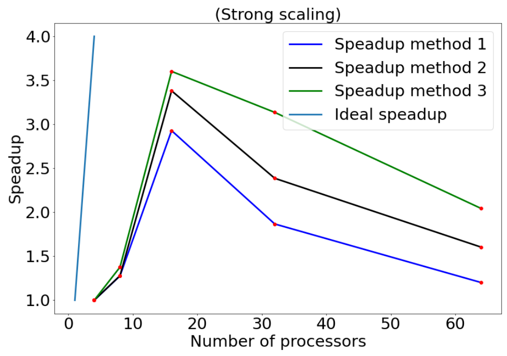

Figure 2: Plots of the speedup of different pivot methods as function of processors' number. The plots are taken for an array ordered backward.

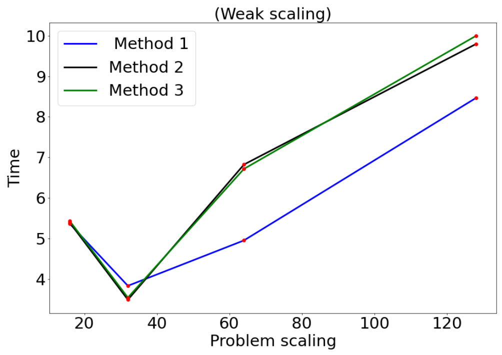

Figure 3: Plot of the clock wall time as a function of the problem size for different pivot methods