Parallel and Distributed Programming
Uppsala University – Spring 2020
Report for Assignment 3: The Parallel Quicksort algorithm

Li Ju, Shuyi Qin

5th June 2020

I Introduction

1 Background

Quicksort is an efficient sorting algorithm. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heapsort.

Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.
Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. Efficient implementations of Quicksort are not a stable sort, meaning that the relative order of equal sort items is not preserved.
Mathematical analysis of quicksort shows that, on average, the algorithm takes O($nlogn$) comparisons to sort n items. In the worst case, it makes O(n^2) comparisons, though this behavior is rare. [1]
In this assignment, a sequence of n (assume that the number of processes is equal to 2^k for some non-negative integer k) integers that needs to be sorted is given. The program is able to performing the Quicksort algorithm in a distributed memory parallel computer environment using C and MPI.

2 Error Control

In this program, all the numbers in the sequences are represented with integer format in C.

3 Computer details

For assignment 3, all tests are conducted on UPPMAX system Rackham. The detailed information of Rackham cluster is listed below:

a) Rackham comprises 9720 cores in the form of 486 nodes with two 10-core Intel Xeon V4 CPU's each. There are "fat" nodes,4 with 1TB memory and 32 with 256 GB memory, with the rest having 128 GB.

b) Rackham's storage system (named Crex) uses the Lustre file system and provides 6.6PB of storage.

c) The interconnect is Infiniband FDR which supports a theoretical bandwidth of 56Gb/s and a latency of 0.7 microseconds.

d) Each compute node consists of two (2) Intel Xeon E5 2630 v4 at 2.20 GHz/core (10 cores, 20 threads, 25 MB LLC, and a bandwidth of 68.3 GB/s) [2]. Each compute node has either 8x16384MB (128GB) or 16x16384MB (256GB) or 16x65536MB (1TB) of ECC 2400MHz DIMM DRAM memory. For local storage each compute node has a 2TB disk at 7200RPM.

e) Each system node also has two CPUs identical to a compute node. System nodes has 16x16384 (256GB) of ECC 2400MHz DIMM DRAM. For local storage each system node contains 4x600GB 10 000 RPM SAS disks for a total of 2.4 TB.

II Solution

1 Algorithm design

For serial implementation, three main steps are required, including data read, sort process and data write. For parallelization, the loop should be parallelized while data read and write will be handled by process 0. The algorithm is shown as Algorithm: parallel implementation.

Algorithm Parallel implementation

Require: Given number of processes N, determine the dimension of hypercube. Partition the sequence equally across dimensions
Sort the sequence locally on each processor
for step=1; step < number of dimension of core-grid; step++ **do**
  Select pivot value and broadcast from processor 0
  Each processor splits locally
  Find the target processor, trades halves across highest dimension
  Sort newly-received data in each processor
  Split sub-communicator forming the two halves
end for
Determine the local data order of each processor
Collect all local result in right order and store it

In the process, three kinds of communication are used, including Ssend (Recv), Isend (Irecv) and Barrier.

1. For initial task (data) assigning, Send (Recv) is used, because the transmission structure is simple, which are one for all and all for one, respectively. The structure is not easily to

be deadlocked, which only requires a process sends (waits for receiving) data from other processes while others only focus on receiving (sending) data. Therefore, Send is used here to simplified the data communication.

2. For communication with neighbors to exchange adjacent values, Isend (Irecv) is used, which is to configure all transmissions first and finally wait for them all finish by another MPI_Wait function. In this case, two advantages arise comparing with Send (Recv), 1). It is not required to wait for neighbor data transmission finish before one process send its own data, which will make the whole communication serial and costly, correspondingly; 2). Each process are in equal position in this scenario and it is easily to get deadlocked. To avoid deadlock, one must start send first and others wait before sending, which requires more work for coding.

3. MPI_Comm_split is used to split the higher dimensional hypercube into lower dimensional cubes. Since the coordinates are binary, we indicate the coordinate of the first dimension as inference to divide the processors into two parts. For example, the processor with 0 as its first dimension coordinate should send the larger halves to those whose coordinate is 1 and receive the smaller halves, vice versa. MPI_Cart_rank is applied to indicate the target processor for exchange data.

4. The coordinate of each processor can be regarded as a binary number that represent the sequence order in each processor. After determining the order of each sorted data group, use MPI_Gather to gather the order and sequence length contains in each processor. MPI_Gather takes elements from many processes and gathers them to the root process. The elements are ordered by the rank of the process from which they were received. It helps the root process to address the sub-sequences.

2 Performance experiments

For performance evaluation, time analysis is introduced. In our performance test, the counted time included the time cost on local sorting, pivot selection, and sub_communicator split (data reading and writing, task assigning and collecting not included). The maximum time cost of all processes is printed and used to represent the time cost of the parallelized program. All the tests are with optimization flag O3 of GCC

In case of strong scalability, the number of processors is increased while the problem size remains constant. This results in a reduced workload per processor. In our experiments, the problem size is fixed by input sequence whose length is 250000000. Each test case is implemented 5 times and the minimum time cost is used to present the time cost of the test case. Numbers of processors are chosen as 1, 2, 4, 8, and 16.

In case of weak scalability, both the number of processors and the problem size are increased. This results in a constant workload per processor. In our experiments, the workload for each processor is fixed as applying sort on the 125000000 number totally. As strong scalability tests, each test case is also implemented 5 times and the minimum time cost is used to present the

time cost of the test case. The detailed test cases are listed in performance analysis part.

III Performance analysis

1 Strong scalability analysis

The results of strong scalability evaluation are listed as Table(1):

Sequence length	pivot	Num of Processors	Time cost/s	Speedup
250000000	1	1	46.969333	1
250000000	1	2	34.344462	1.368
250000000	1	4	20.104709	2.336
250000000	1	8	12.184565	3.855
250000000	1	16	7.822283	6.005

Table 1: Strong scalability results

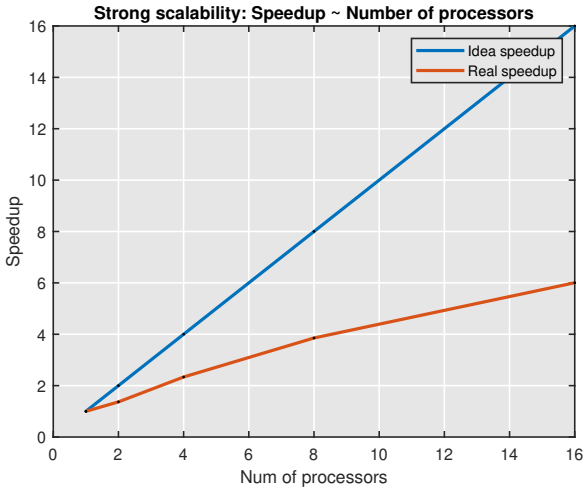

Figure 1: Strong scalability: speedup versus number of processors

With the data, a figure of speedup versus number of processors is plotted as Figure (1) shows. When the amount of task is fixed, as the number of processors increases, the speedup of the program increase correspondingly. The entire speedup line is lower than idea speedup. As the number of processors increases, the time cost on message passing and barrier waiting increases significantly, while the time cost on calculation is keeping reducing. When the number of processors is small (1 and 2), most of the running time cost on calculation part. But when the processor number increases, most time costs on pivot selection, communicator splitting, message passing and barrier waiting. Though the calculation time keeps reducing, the reduction of time cost has less impact compare with the increase of communication cost. Therefore, after the peak, the speedup is reducing.

Serial quicksort is well known to work well in the average cases. But in some scenario, for example, all elements of the small array are less than the median of the larger array or all element of the smaller array are greater than the median of the larger array, the performance of the serial quicksort can be really bad. To figure out the impact of the sequence balance, further tests on this question is done. To check how much the sequence itself would affect the performance, time evaluation for a backward sequences is done and the results are shown as Table(2).

With the data, it is obvious that when the amount of task is fixed, the time cost of sorting the

Num of Processors	pivot	sequence length	random	backward
1	1	125000000	20.937312	7.732412
2	1	125000000	15.545540	7.755649
4	1	125000000	10.090057	9.123227
8	1	125000000	5.520625	5.744821
16	1	125000000	3.639584	4.289862

Table 2: The compare between random and backward sequence

backward sequence is much lower than the random sequence.

2 Weak scalability analysis

The results of weak scalability evaluation are listed as Table(3):

Num of Processors	Num of elements	pivot	Time cost/s	Expected cost/s	Efficiency
1	125000000	1	20.937312	-	1
2	125000000	1	34.344462	20.937312	0.6096
4	125000000	1	40.344326	20.937312	0.5190
8	125000000	1	52.000819	20.937312	0.4026
16	125000000	1	69.824470	20.937312	0.2999

Table 3: Weak scalability: results

The data is illustrated graphically by Figure(2). Ideally, when the workload of each core is fixed, the efficiency should always be 1 when the number of core increases. However here in this case, the efficiency decreases. The reason of this can be explained by what we found in strong scalability part: when the number of processors increase, the time cost on pivot selection, communicator splitting, inter-core communication and barrier waiting increases dramatically, which make the efficiency decrease. As we pointed out, when the number of cores reaches 16, most of total running time are cost on the two parts and the calculation time cost is relatively small.

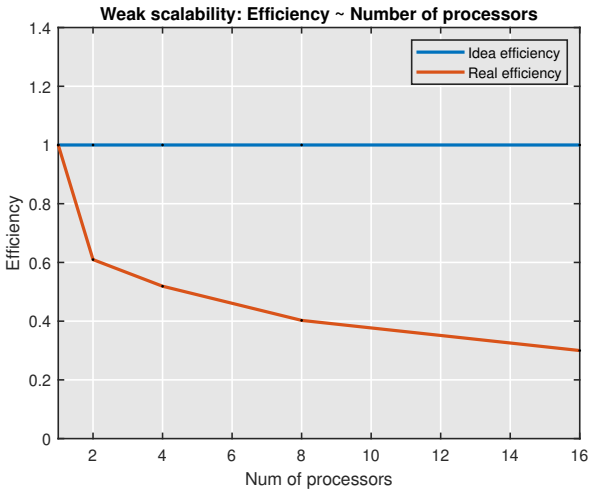

Figure 2: Weak scalability: efficiency versus number of cores

To figure out the impact of different pivot selection strategy, further time analysis is done. The results of performance evaluations using different pivot selection strategy are listed as Table(4):

Sequence length	Processors	Speedup (strategy 1)	Speedup(strategy 2)	Speedup(strategy 3)
250000000	1	1	1	1
250000000	2	1.3676	1.7055	1.4714
250000000	4	2.3362	2.8493	1.9799
250000000	8	3.8548	4.2953	1.4361
250000000	16	6.0046	6.3011	1.5010

Table 4: Strong scalability results with different pivot

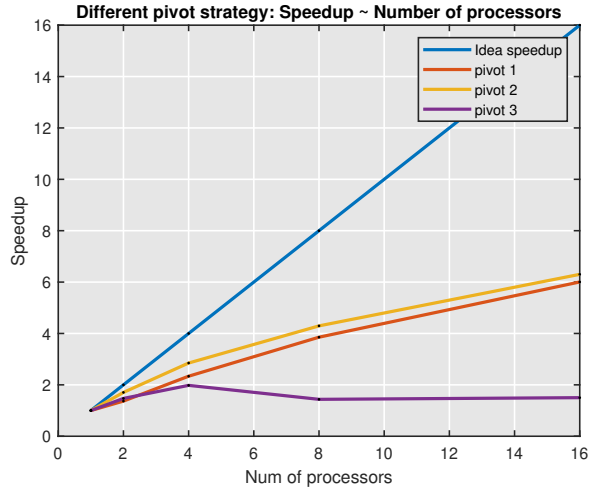

Figure 3: Strong scalability: speedup versus number of processors

The performance of strategy 2 which select the median of all medians in each processor group

is better than the others. The sequences can be segmented more balanced using this strategy compare with the strategy 1, and spend less time than strategy 3.

IV Conclusion

In the assignment, the quicksort algorithm is parallelized with MPI. Performance analysis are conducted, and based on the performance results, more detailed tests and discussions are made.

References

[1] Wikipedia, Quicksort, `https://en.wikipedia.org/wiki/Quicksort`.

[2] Xin Li,Scalability: strong and weak scaling, `https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/`.