Report – Assignment 2

Jiayi Yang (Group 37)

1 Introduction

Given a sequence of *n* integers, the task is to implement the Parallel Quicksort algorithm using C and MPI, and evaluate the performance.

The Parallel Quicksort algorithm is as below:

1 Divide the data into p equal parts, one per process

2 Sort the data locally for each process

3 Perform global sort

3.1 Select pivot element within each process set

3.2 Locally in each process, divide the data into two sets according to the pivot (smaller or larger)

3.3 Split the processes into two groups and exchange data pairwise between them so that all processes in one group get data less than the pivot and the others get data larger than the pivot.

3.4 Merge the two sets of numbers in each process into one sorted list

4 Repeat 3.1 - 3.4 recursively for each half until each group consists of one single process.

There are three pivot strategies:

1. Select the median in one processor in each group of processors.

2. Select the median of all medians in each processor group.

3. Select the mean value of all medians in each processor group.

2 Implementation

In my implementation, the master processor read the data from file and store it in an array, then calculate the size of each chunk that each processor will contain. After broadcasting the size and each processor creates an empty array to contain the chunk of data, the master processor scatters the data to the processors. Then each processor sort its own chunk in ascending order. Then in each step, the master processor gathers the median in each processor and calculate the pivot by the type users given. In each processor group, the processors are divided into two equal parts, one part will contain the numbers smaller than pivot, while the other part will contain the numbers larger than pivot. The smaller part send the larger numbers in its chunk to the larger part and vice versa. Then each processor sorts its own chunk, and each part become a new group then get into next step, until each group only have one processor. The master processor gathers the data in the chunks and write into file.

3 Numerical experiments

First, run the program on the file contain 10 numbers to confirm the correctness of the program, with all the 3 types.

Then run the program on 125000000 numbers with 1, 2, 4, 8 and 16 cores to demonstrate the strong scalability, and 250000000 numbers with 2 cores, 500000000 numbers with 4 cores, 1000000000 numbers with 8 cores and 2000000000 numbers with 16 cores to demonstrate weak scalability. Also, run the program on 125000000 and 2000000000 descending order numbers with 16 cores.

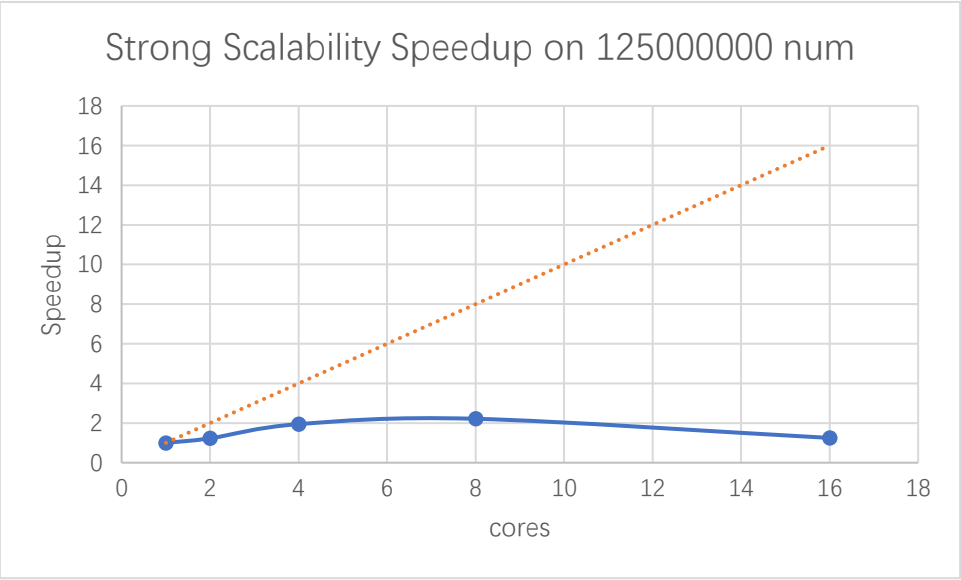
All of the numerical experiments are ran at Rackham on UPPMAX.

4 Result

1) Strong scalability

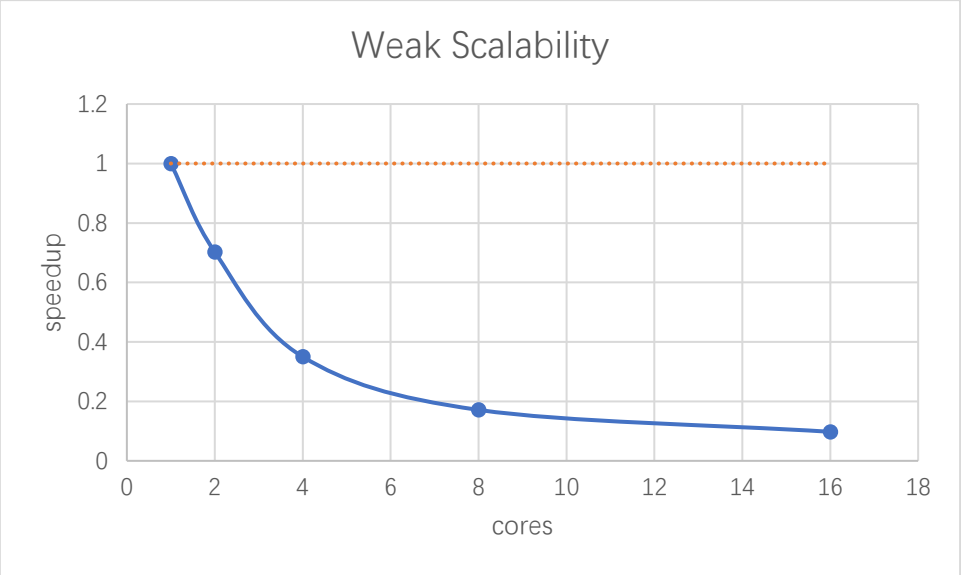
Time(s) on 125000000 numbers with different number of cores

cores	type			Speedup with type 1
	1	2	3	
1	21.454803			1
2	17.428052	17.474867	17.675442	1.23105
4	11.026230	15.465406	15.483258	1.945797
8	9.683487	13.633531	16.086454	2.215607
16	17.158281	17.909967	17.744272	1.250405



2) Weak scalability

cores	n	type			Speedup with type 1
		1	2	3	
1	125000000	21.454803			1
2	250000000	30.543514	37.715212	37.269436	0.702434
4	500000000	61.279199	67.593091	68.789321	0.350116
8	1000000000	125.182778	141.254513	117.117944	0.171388
16	2000000000	219.551590	313.555577	286.082639	0.097721



3) Descending order numbers

cores	n	Backwards or input	Type		
			1	2	3
16	125000000	backwards	15.767436	15.742068	15.752429
		input	17.907143	18.332607	18.224794
16	2000000000	backwards	269.599033	208.685872	218.005495
		input	219.551590	313.555577	286.082639

5 Discussion

Either the strong scalability or the weak scalability of this program is not good. I conjecture the reasons are as follow:

1) More cores cause to more overhead, if the time spending on communication between the cores is quite longer than the calculation time in each core, it will not have a good speedup.

2) According to the algorithm, for p processors, it will converge in $\log_2 p$ steps, which means, each processor should sort sequences $\log_2 p$ times, it may take more time.

3) By tracking the size of the chunk in each processor, different processor has very different amounts of numbers to handle. The best condition is each processor always contains n/p numbers as in the very first step, but this hardly happens. In one step, some processors may only need to sort hundreds of numbers, while some other processors have a heavy job to do, so some processers finish their job quickly then wait for the others. Therefore, sometime only a few processors are working, which is the most important reason in my opinion.

   We can see different type of choosing pivot may cause very different runtime, that is because the rule of choosing pivot may affect the size of smaller and larger part in one chunk, then cause the different of chunk size in next step. This also can explain why sorting descending order sequences of numbers usually takes less time than the random ones. When dealing with backwards numbers, each processor handles numbers which are closed to each other, so after sending and receiving, processors still have similar size of chunk, the problem can be solved mostly in a parallel way.

   In conclusion, improper pivot may cause the machines cannot unleash their full potential ability. Therefore, finding a better rule of choosing pivot may improve the performance of the program.