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.

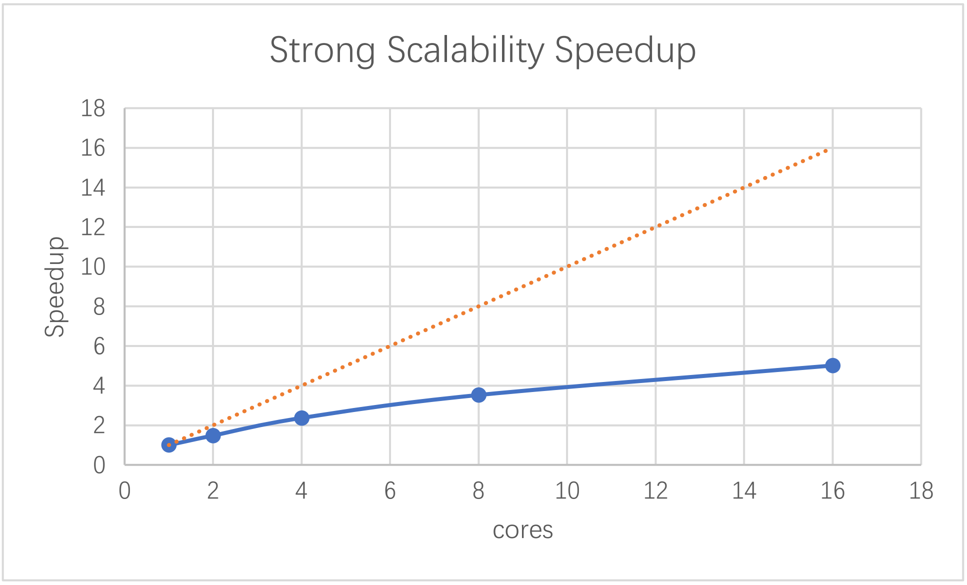
All of the numerical experiments are ran at Rackham on UPPMAX.

4 Result

1) Strong scalability

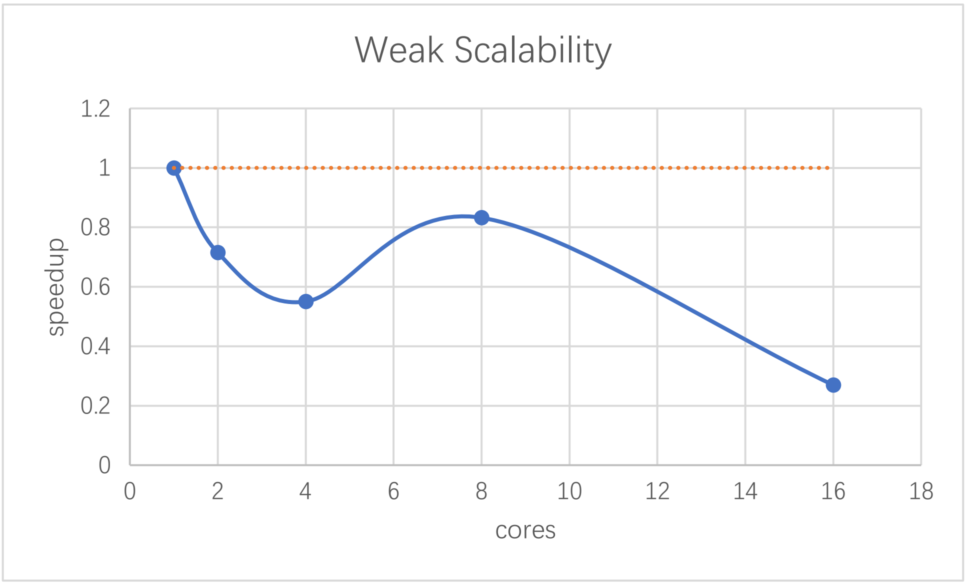
Time(s) on 500000000 numbers with different number of cores

Cores	Time(s)	Speedup
1	91.88211	1
2	62.07634	1.480
4	38.822456	2.366
8	26.059356	3.525
16	18.327846	5.013



2) Weak scalability

Cores	N	Time(s)	Speedup
1	125000000	21.366332	1
2	250000000	29.887945	0.714
4	500000000	38.822456	0.550
8	1000000000	25.673354	0.832
16	2000000000	79.259693	0.269



3) Descending order numbers and the effect of different pivot strategy

cores	n	Backwards or input	Type		
			1	2	3
16	125000000	backwards	5.013832	3.985586	3.312837
		input	4.324880	4.336444	4.342167
16	2000000000	backwards	90.398380	71.752130	60.347743
		input	79.223528	79.259693	79.245657

5 Discussion

The program has good weak scalability with less cores, but the strong scalability is not good enough. 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, for example, use 16 cores to sort only 100 numbers, it will not have a good speedup.

2) By tracking the time for each part of my implement, for example, for 500000000

numbers with 16 cores, I found that, in total it takes 18 seconds, the sorting part in each processor in each step takes 1.8 seconds, and total time in each step is 2.2 seconds, where it have 4 steps, and the whole merge steps takes 10 seconds. That is reasonable to some extent, and in each step, it needs to allocate memory, store data, free memory and something else, which may take some time and maybe could be optimized. But in the very first part, that the master processor scatters the data to all the processors, then the processors sort locally for the first time, this part takes 8 seconds, which is out of my expectation. I guess the reason is the processors didn't create local array and get the data at the same time, so some processors must to wait others completing the sort part, then they can go to the merge steps.

3) We can see different type of choosing pivot may cause very different runtime for descending order sequences, but doesn't have such effect on random sequences. 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. For random sequences, medians from each processor are similar, so the pivots calculated by different strategy may not have huge difference. But for descending order sequences, numbers in each processor are similar, so the medians from the processors may be different with each other, the pivot strategy may make sense. Here we can see, the mean of the medians has the best performance, whereas choose median in one processor represent the group's pivot is the worst, because median in one processor can hardly fit numbers in other processors.