Parallelization of Quicksort

Victoria Sedig
Parallel and Distributed Programming
Uppsala University

26 May 2021

1 Introduction

1.1 Background

In this project a parallelized version of quicksort will be implemented. The lists to be sorted will be of type double.

2 Method

2.1 Parallelization

Processor zero (root) will create the data with a determined length. After the data generation processor zero will scatter values for each processor. All processors will then quicksort their list so that every processor has a sorted list. Then a recursive algorithm will start. In the algorithm a pivot element will be chosen. The processors will divide their data into two groups (larger and smaller than the pivot). After that the processor are divided into two groups. Then the processors will send their lists to the other group pairwise. One group will send all their smaller lists and one group will send all their larger lists. This is repeated until the groups of processors it only one processor big. The number of processors are assumed to 2^k where k is a positive integer. The lists from each processor is then gathered to process 0 which will give the final sorted list.

2.2 Random data generation

How process zero will generate the random values to fill the data is different depending on what the user gives as input. The options are as following:

- input=0 $\rightarrow$ Uniform random numbers
- input=1 $\rightarrow$ Exponential distribution
- input=2 $\rightarrow$ Normal distribution
- input=3 $\rightarrow$ Descending order

.

2.3 Pivot generation

There are three different options for how the pivot number is chosen. The options are the following:

- input=0 → One processors median
- input=1 → The median off all medians in group
- input=2 → The mean value off all medians in group

2.4 Verification of Correctness

To verify the correctness the results were compared with the results from the sequential given quicksort algorithm.

3 Experiments

The tests were run with one bash file on the snowy cluster. There were tests with growing numbers of processors where (to give speedup). It was done for all combinations of random number generations and pivot strategies. The same thing was done but with rising number of data and number of processors so that the work per processor was kept were constant.

4 Results

4.1 Strong Scalability

Table 1-4 are the results from the strong scalability experiments where each table has a different random sequence generation method. The tables include variation of pivot method option and the length of the list is constant.

Table 1: The time and speedup for increasing numbers of processors and different pivot methods. The length of the list was constantly $2 * 10^8$. The speedup was calculated with the sequential time $t_{seq} = 26.640941$. This table has the results for when the random sequence generation was option 0.

Pivot-Method	**#Cores:**	**2**	**4**	**8**	**16**	**32**
0	Time	15.7546	9.4762	5.7540	3.9472	2.9591
	Speedup	1.6910	2.8113	4.6300	6.7493	9.0032
1	Time	15.7166	9.4842	5.6661	3.9496	2.9863
	Speedup	1.6951	2.8090	4.7018	6.7452	8.9210
2	Time	15.7783	9.4739	5.7659	3.9435	3.0370
	Speedup	1.6885	2.8120	4.6204	6.7557	8.7722

Table 2: The time and speedup for increasing numbers of processors and different pivot methods. The length of the list was constantly $2 * 10^8$. The speedup was calculated with the sequential time $t_{seq} = 26.506582$. This table has the results for when the random sequence generation was option 1.

Pivot-Method	**#Cores:**	**2**	**4**	**8**	**16**	**32**
0	Time	15.0031	8.9275	5.5272	3.8682	2.9735
	Speedup	1.7667	2.9691	4.7957	6.8525	8.9143
1	Time	14.9244	8.9103	5.6221	3.9175	2.9593
	Speedup	1.7761	2.9748	4.7147	6.7662	8.9570
2	Time	14.9207	8.9283	5.6629	3.8999	3.0031
	Speedup	1.7765	2.9688	4.6808	6.7968	8.8265

Table 3: The time and speedup for increasing numbers of processors and different pivot methods. The length of the list was constantly $2 * 10^8$. The speedup was calculated with the sequential time $t_{seq} = 26.897380$. This table has the results for when the random sequence generation was option 2.

Pivot-Method	**#Cores:**	**2**	**4**	**8**	**16**	**32**
0	Time	15.1374	8.5769	5.3572	3.7158	2.9730
	Speedup	1.7769	3.1360	5.0208	7.2387	9.0472
1	Time	14.9495	8.5915	5.3502	3.7251	2.9805
	Speedup	1.7992	3.1307	5.0273	7.2205	9.0244
2	Time	14.9847	8.5613	5.3533	3.7249	3.0211
	Speedup	1.7950	3.1418	5.0244	7.2210	8.9032

Table 4: The time and speedup for increasing numbers of processors and different pivot methods. The length of the list was constantly $2 * 10^8$. The speedup was calculated with the sequential time $t_{seq} = 8.149103$. This table has the results for when the random sequence generation was option 3.

Pivot-Method	**#Cores:**	**2**	**4**	**8**	**16**	**32**
0	Time	7.1625	6.0828	4.5585	3.8275	3.0316
	Speedup	1.1378	1.3397	1.7877	2.1291	2.6880
1	Time	6.6487	5.1843	3.8294	3.0928	2.5376
	Speedup	1.2257	1.5719	2.1280	2.6349	3.2113
2	Time	6.6624	5.2154	4.0374	3.0897	2.4976
	Speedup	1.2232	1.5625	2.0184	2.6375	3.2627

In figure 1 the results from the strong scalability is shown. Each subplot represents a different random sequence generation option.

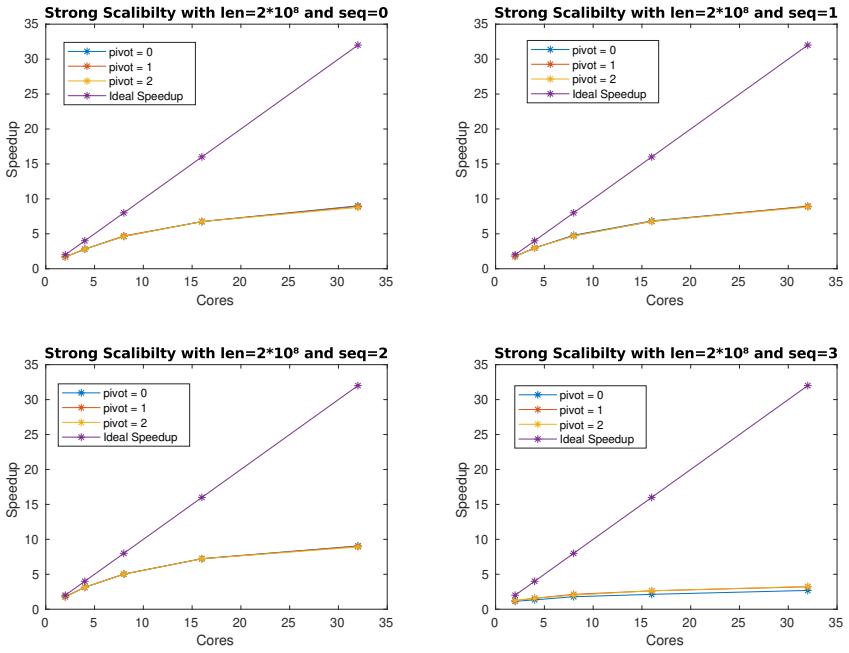

Figure 1: The speedup and the ideal speedup for the parallelized program plotted against the number of cores used for different pivot methods. The length of the list was kept constant at $len = 2^8$.

4.2 Weak Scalability

Table 5-8 are the results from the weak scalability experiments where each table has a different random sequence method. The tables include variation of pivot method option and the number of processors and length of list increases.

Table 5: The weak scalability for different pivot methods. The length of the list was increased so that each processor always have around the same amount of work ($125 * 10^6$ elements). This table has the results for when the random sequence generation was option 0.

#Cores	1	2	4	8	16
#Elements	125*10^6	250*10^6	500*10^6	100*10^7	200*10^7
Time Pivot = 0	16.7052	20.0648	23.1385	30.1958	39.1223
Time Pivot = 1	16.7052	20.0379	23.2536	29.8326	39.1655
Time Pivot = 2	16.7052	19.9772	22.9633	29.9300	39.1130

Table 6: The weak scalability for different pivot methods. The length of the list was increased so that each processor always have around the same amount of work ($125 * 10^6$ elements). This table has the results for when the random sequence generation was option 1.

#Cores	1	2	4	8	16
#Elements	125*10^6	250*10^6	500*10^6	100*10^7	200*10^7
Time Pivot = 0	16.4088	19.1711	23.0117	28.3364	39.2869
Time Pivot = 1	16.4088	19.2018	22.4537	28.2343	38.8570
Time Pivot = 2	16.4088	19.2130	22.6709	28.3032	38.8877

Table 7: The weak scalability for different pivot methods. The length of the list was increased so that each processor always have around the same amount of work ($125 * 10^6$ elements). This table has the results for when the random sequence generation was option 2.

#Cores	1	2	4	8	16
#Elements	125*10^6	250*10^6	500*10^6	100*10^7	200*10^7
Time Pivot = 0	16.5970	19.2033	22.6403	27.7988	40.0581
Time Pivot = 1	16.5970	19.1967	22.4280	28.5814	38.6237
Time Pivot = 2	16.5970	19.1780	22.4469	28.2084	38.8585

Table 8: The weak scalability for different pivot methods. The length of the list was increased so that each processor always have around the same amount of work ($125 * 10^6$ elements). This table has the results for when the random sequence generation was option 3.

#Cores	1	2	4	8	16
#Elements	125*10^6	250*10^6	500*10^6	100*10^7	200*10^7
Time Pivot = 0	5.0966	9.0481	13.8017	22.3889	31.9664
Time Pivot = 1	5.0966	8.2520	11.3445	16.9381	25.9890
Time Pivot = 2	5.0966	8.2445	11.3782	16.8354	26.6473

In figure 1 the results from the weak scalability is shown. Each subplot represents a different random sequence generation option.

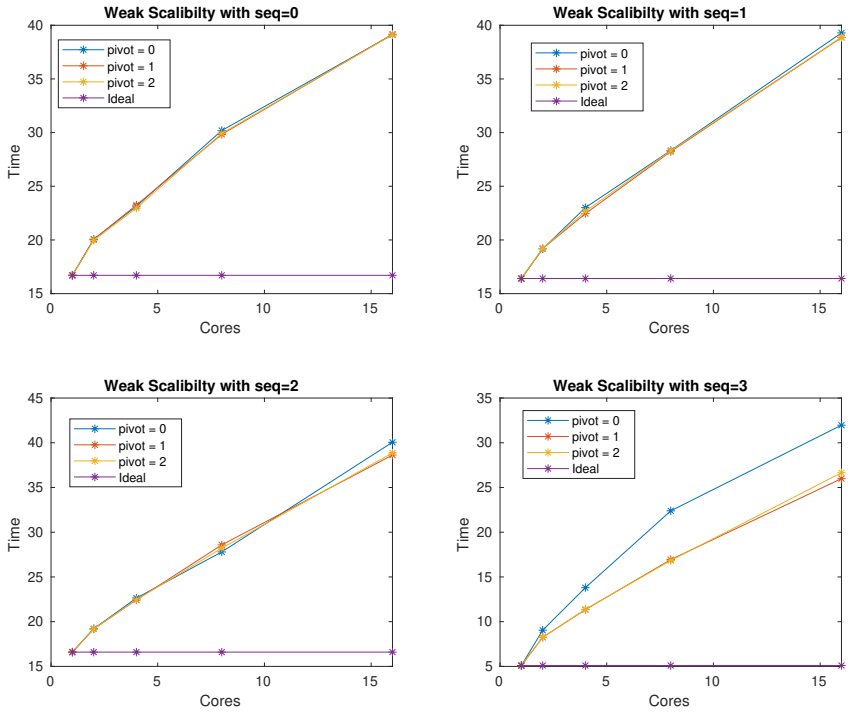

Figure 2: The execution time for the sequential and parallelized code plotted against length of the list with different pivot methods. The number of processors were kept constant at 16.

5 Discussion

In figure 1 the strong scalability for different random sequence generation and for different pivot methods is shown. Different pivot methods do not seem to effect the speedup since the different pivot method on each subplot are very similar. The pivot option 0 has the worst speedup for the sequence generation option 3. That is reasonable since pivot option 0 only chooses pivot based on process with rank zero in the group and the sequence is in descending order. That means that the other processes either have to send or keep their whole array (since all of their values are smaller than the pivot but only half the group will send their small elements). This will make the distribution of elements unbalanced which could be what has affected the

performance.

The different sequence generation method affect the result more than the pivot methods. According to figure 1 the best speedup is for option 2 (normal distribution) which is for most number of processors slightly better than option 0 and 1. The options 0-2 are still very similar. Sequence generation option 3 on the other hand has low speedup. However if table 4 is compared to table 1-3 it is seen that the times are generally a lot lower for table 4. The reason the speedup is bad may be because the descending order is an optimal case for the sequential program. The sequential time for sequence generation 3 is almost three times faster than the the other sequential times meanwhile the parallelized code is only around two times faster for the sequence generation 3.

In figure 2 the weak scalability for different random sequence generation and for different pivot methods is shown. For sequence generation option 0-2 neither the sequence generation or pivot method makes a significant difference between the plots. Sequence generation option 3 on the other hand is both faster and with a change between pivot methods. It is the pivot method 0 that performs slightly worse than the others. It may be because of the same reason as was already discussed for figure 1.