Parallel Quicksort

David Hovstadius
Oskar Lernholt
Gustav Malmsten

UPPSALA
UNIVERSITET

Parallel and distributed programming (1TD070)
Teknisk fysik, Uppsala universitet
May 7, 2023

1 Introduction

The quicksort algorithm consists of recursively partitioning a list into sub-lists, one consisting only of the elements smaller than a pivot element, and one consisting of the elements that are larger than or equal to the same pivot element. The speed of this algorithm is dependent of the choice of pivot element in combination with the distribution of the elements to be sorted. Three different pivot strategies will therefore be tested both on uniformly distributed random data, and data in descending order. The pivot strategies investigated in this paper are the following:

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.

To parallelise this algorithm, one may use algorithm:

Algorithm 1 Parallel Quicksort

1. Divide the data into p equal parts, one per process
2. Sort the data locally for each process
3. Perform global sort
    1 Select pivot within each processor set
    2 Locally in each process, divide the data into two sets according to the pivot (smaller or larger)
    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.
    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.

2 Description of implementation

The implementation of this algorithm uses a distributed memory environment in which the root process (0) handles the reading and writing to the input and output

files. After the root process has read and stored the contents of the input file, it distributes sub lists of equal sizes to all $p-1$ processes using MPI_Scatterv, and lets the last process get a larger chunk of numbers, including the remainder of N/p. Thereafter, each process locally sorts it's sublist using the function qsort from the C standard library. Then, the global sort begins which consists of $\log_2 p$ iterations. In each iteration, the local *communicator* broadcasts the pivot value to all other processes in the processor group followed by the processes splitting it's sub list into two, one with numbers smaller than the pivot value and one with numbers greater than or equal to the same. The communicators are the root processes in each local MPI communicator, called MPI_LOCAL_COMM in our implementation. The processes are then split into two groups, those with rank lower than $p/2$ and those with rank higher than or equal to $p/2$. The sub lists are then exchanged in accordance with step *3.3* in algorithm 1 using a nonblocking send (MPI_Isend) followed by a blocking receive (MPI_Recv). Therafter, the two sorted sub lists are merged into one large, sorted list.

After this, the local communicator is split into two new, groupwise like described in step 3.3 in the algorithm.

After the $\log_2(p)$ iterations, the chunks and displacements are finally updated before all sub lists are gathered into the global list by the root process using MPI_Gatherv.

3 Experiments

The experiments were run on Uppsala University Linux machine Vitsippa (CPU: AMD Opteron (Bulldozer) 6282SE, 2.6 GHz, 16-core, dual socket) For the strong scaling experiments, two experiments with fixed array sizes of 125000000 and 250000000 numbers will be conducted. For the weak scaling experiments, the follwing combinations of number of processes p and number of elements N will be used:

p	N
1	125000000
2	250000000
4	500000000
8	1000000000
16	2000000000

Table 1: Caption

In all experiments, all three different pivot strategies will be tried.

3.1 Weak scaling

3.1.1 Random data

Table 2: Weak scaling using pivot strategy 1

Data Size	Number of Processors	Normalised time
125000000	1	1.000000
250000000	2	1.273215
500000000	4	1.070928
1000000000	8	1.403479
2000000000	16	1.658289

Table 3: Weak scaling using pivot strategy 2

Data Size	Number of Processors	Normalised time
125000000	1	1.000000
250000000	2	1.262585
500000000	4	1.150717
1000000000	8	1.418834
2000000000	16	1.641507

Table 4: Weak scaling using pivot strategy 3

Data Size	Number of Processors	Normalised time
125000000	1	1.000000
250000000	2	1.891949
500000000	4	1.144112
1000000000	8	1.199881
2000000000	16	1.550030

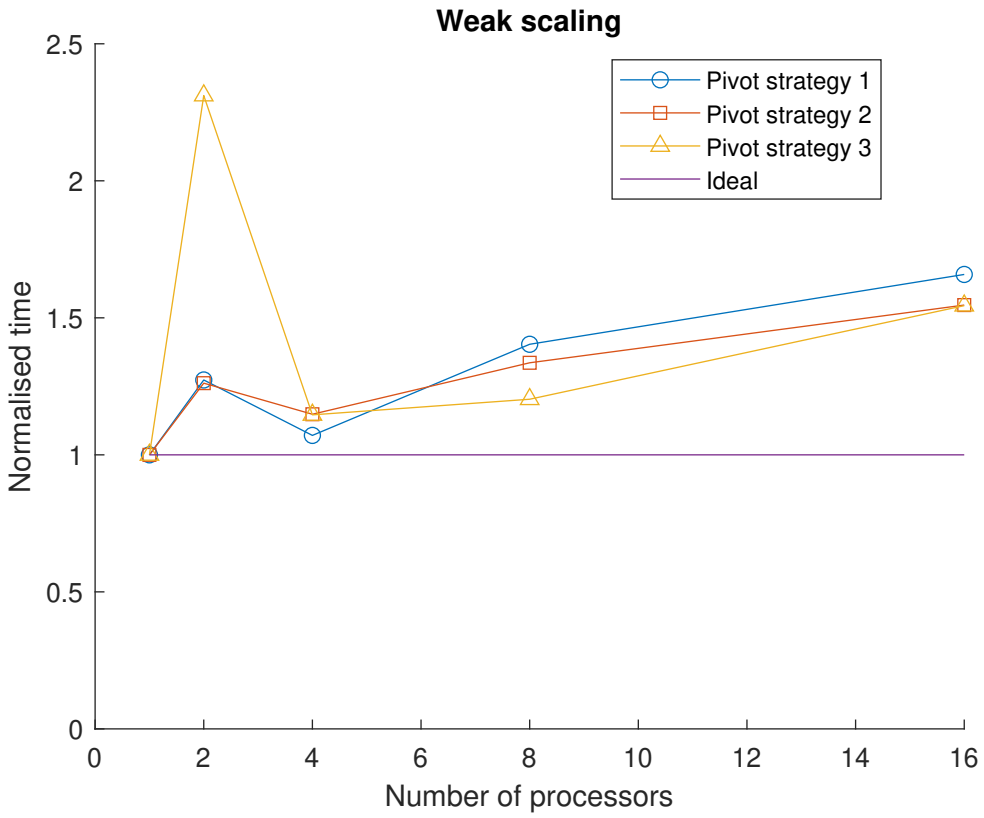

Figure 1: Weak scaling for different choices of pivot strategy on uniformly distributed random data

3.1.2 Data in descending order

Table 5: Weak scaling using pivot strategy 1

Data Size	Number of Processors	Normalised time
125000000	1	1.000000
250000000	2	1.775398
500000000	4	1.317528
1000000000	8	1.618467
2000000000	16	2.425698

Table 6: Weak scaling using pivot strategy 2

Data Size	Number of Processors	Normalised time
125000000	1	1.000000
250000000	2	2.014931
500000000	4	1.215916
1000000000	8	1.677090
2000000000	16	2.205033

Table 7: Weak scaling using pivot strategy 3

Data Size	Number of Processors	Normalised time
125000000	1	1.000000
250000000	2	1.851180
500000000	4	1.319000
1000000000	8	1.827996
2000000000	16	3.693898

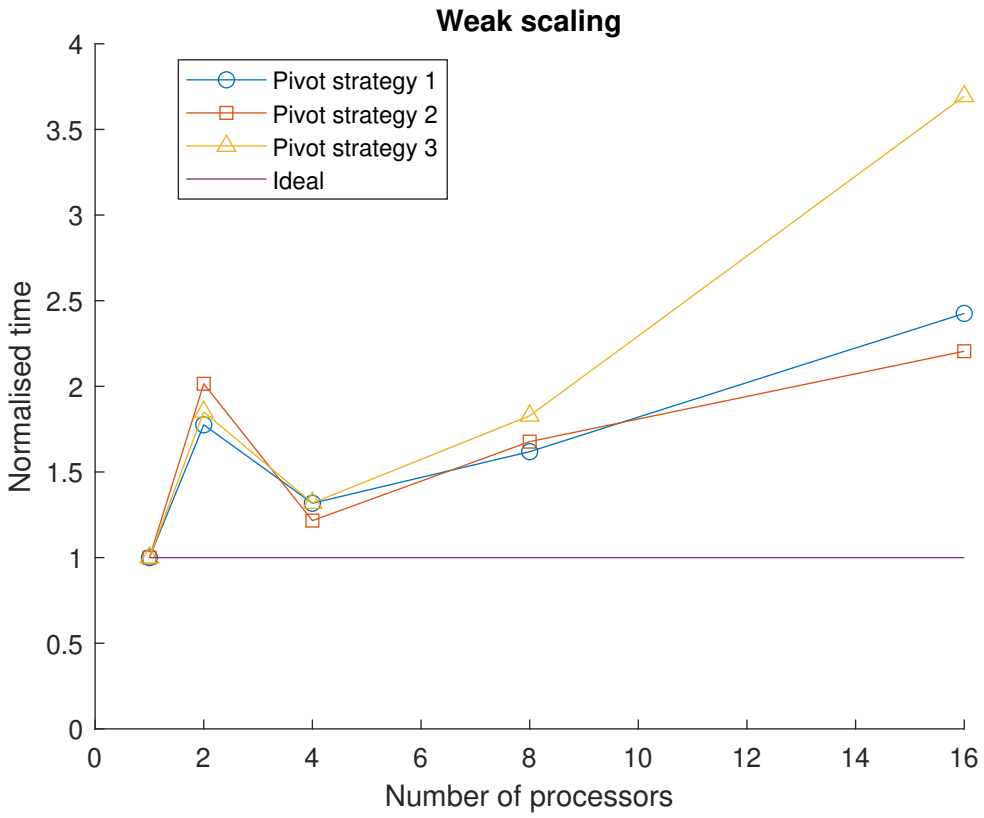

Figure 2: Weak scaling on array with data in descending order for different pivot strategies

3.2 Strong scaling

3.2.1 Random data

Table 8: Strong scaling using pivot strategy 1

Data Size	Number of Processors	Speedup
250000000	1	1.000000
250000000	2	0.897938
250000000	4	3.245926
250000000	8	5.370632
250000000	16	7.942547

Table 9: Strong scaling using pivot strategy 2

Data Size	Number of Processors	Speedup
250000000	1	1.000000
250000000	2	0.946811
250000000	4	2.777586
250000000	8	5.381842
250000000	16	7.924297

Table 10: Strong scaling using pivot strategy 3

Data Size	Number of Processors	Speedup
250000000	1	1.000000
250000000	2	0.959407
250000000	4	2.910278
250000000	8	6.289914
250000000	16	7.912491

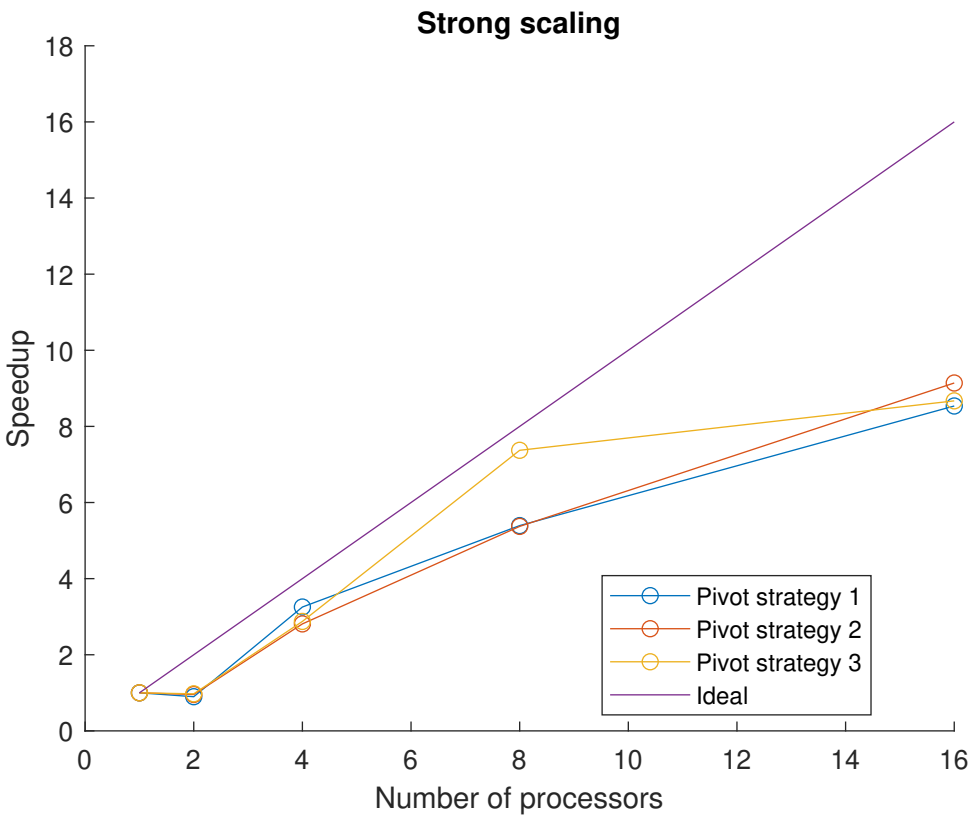

Figure 3: Strong scaling for different choices of pivot strategy on uniformly distributed random data

3.2.2 Data in descending order

Table 11: Strong scaling using pivot strategy 1

Data Size	Number of Processors	Speedup
250000000	1	1.000000
250000000	2	1.092667
250000000	4	5.384978
250000000	8	8.426514
250000000	16	9.7887669

Table 12: Strong scaling using pivot strategy 2

Data Size	Number of Processors	Speedup
250000000	1	1.000000
250000000	2	1.478020
250000000	4	7.013462
250000000	8	10.532931
250000000	16	13.054325

Table 13: Strong scaling using pivot strategy 3

Data Size	Number of Processors	Speedup
250000000	1	1.000000
250000000	2	1.821742
250000000	4	6.734645
250000000	8	11.405519
250000000	16	14.372640

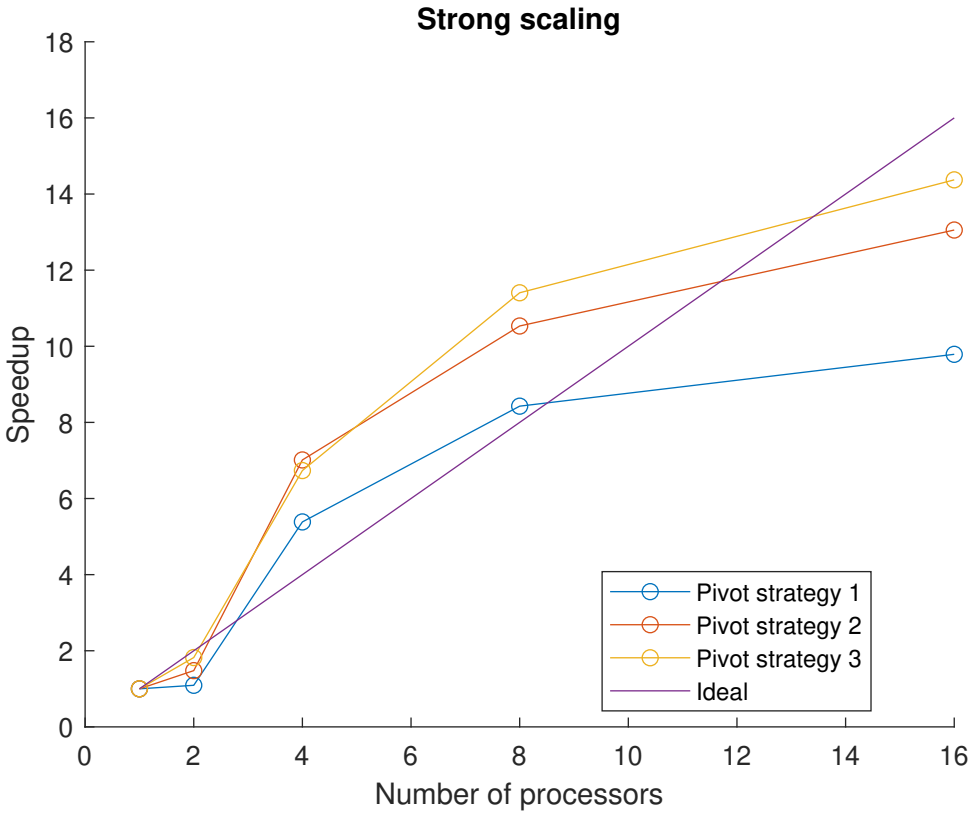

Figure 4: Strong scaling on array with data in descending order for different pivot strategies

4 Discussion

Regarding the experiments in general, the results are very desirable. Both the weak and strong scaling show signs of good performing parallel code.

One outlier that is prominent in almost all the above experiments is the decreased speedup for 2 processors compared to a single processor. The underlying reason is that when the code becomes parallel it adds overhead in the form of iterating, splitting, sending, and receiving data to the grouped processors. The time added for these overheads are greater than what the computational time is for the case of 2 processors and therefore the total run time increases slightly instead of decreasing.

The findings of the weak scaling experiments demonstrate that the program's runtime scales well with increasing numbers of processors, but falls short of ideal scaling behavior. In particular, when executing the program with only two pro-

cesses, the normalized runtime is observed to increase by a factor of approximately two for descending-order data. Similarly, for random data using pivot strategy 3, the normalized runtime is roughly 2.5 times slower than the single process case. This non-ideal behavior is expected since there is communication between the processes for each iteration which adds significant overhead as compared to the non-parallel, non-communicating case.

The results depicted in Figures 1 and 2 indicate that the performance of pivot strategy 3 varies significantly when sorting random versus descending order data. It is postulated that this discrepancy arises due to load balancing issues stemming from the choice of the pivot value.

Both weak and strong scaling experiments exhibit differences in their results depending on whether a random list or a list sorted in descending order is used for sorting. In particular, a reverse sorted list represents the worst-case scenario for quick sorting, with time complexity of $O(N^2)$ instead of the optimal and average time complexity of $O(N \log(N))$. In the case of weak scaling experiments, the data was scaled to maintain an average time complexity, and as such, the performance is adversely affected when sorting the worst-case scenario list.

More specifically, the performance degradation observed for pivot strategy 3 may be attributed to suboptimal load balancing and communication overhead during the parallel sort operation. This is because selecting a suboptimal pivot value may result in an uneven distribution of elements among different processors, leading to load imbalance and poor scaling behavior. Moreover, communication overhead is a significant factor when dealing with a large number of processes, and can further exacerbate load balancing issues, particularly when data movement is not balanced among processors.

Analyzing the strong scaling for uniformly distributed data seen in figure 3, all pivot strategies are close to linear up to 8 processors. After this point all pivot strategies start to decrease going to 16 processors, this is very usual behavior and a result of overhead from the many processors in use.

The reason for the speedup not being perfectly linear is that the number of iterations done is being scaled with log_2 of the number of processors. Therefore it is not possible to reach perfect linear speedup, but close to it.

Looking at the strong scaling for the backwards sorted data seen in figure 4, there are some remarkable results. The speedup for 4 and 8 processors is above ideal for all pivot strategies. This is rare but possible. Reversing a list is usually a computationally complex task for a sorting algorithm, when the algorithm is run with more processors the time for these computations decreases greatly and the overhead increase will become small in comparison that the speedup therefore becomes a little better than ideal. Which seems to be the case for this implemented algorithm.

Analyzing the different pivot strategies. They performed similar in most of

the experiments. For the weak scaling with uniformly distributed data, pivot strategy number 3 seems to perform the worst when evaluating the weak metric for 2 processors compared to the others before they all start to even out for more processors. For the backwards sorted data they all perform similarly until a larger number of processors are being used, then pivot strategy 3 worsens.

For the strong scaling metric when uniformly distributed data is used the choice of pivot strategy does not matter, they performed equally well. And lastly for the backwards sorted data, all pivot strategies performed above ideal with strategy 3 performing the strongest.

To summarize, this implementation of the parallel quicksort algorithm is performing well on both the weak and strong scaling metrics, except for the case of 2 processors. The choice of pivot strategy will effect the total time but not to a huge extent. Further work has to be done to get a better understanding of how the pivot strategy effects the time complexity. The work needs to be done on more distributions of data, some examples are normal and exponential distributions.