Parallel Quicksort

David Hovstadius
Oskar Lernholt
Gustav Malmsten

UPPSALA
UNIVERSITET

Parallel and distributed programming (1TD070)
Teknisk fysik, Uppsala universitet
May 19, 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 UPPMAX computing cluster Snowy. For the strong scaling experiments, two experiments with fixed array sizes of 125000000 and 250000000 numbers will be conducted. The arrays to be sorted are located in the directory /proj/uppmax2023-2-13/nobackup/qsort_indata. 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 investigated.

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

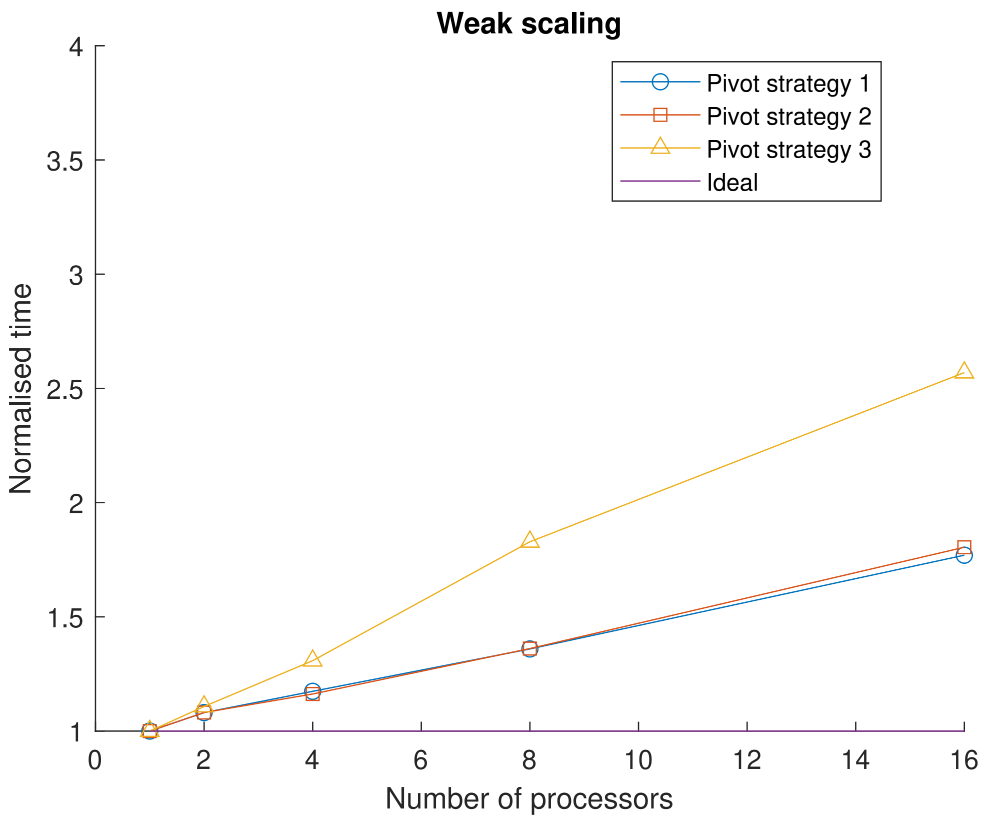


Figure 1: Weak scaling for different choices of pivot strategy on uniformly distributed random data

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	1.188688
500000000	4	1.545999
1000000000	8	2.048663
2000000000	16	3.029716

Table 6: Weak scaling using pivot strategy 2

Data Size	Number of Processors	Normalised time
125000000	1	1.000000
250000000	2	1.188972
500000000	4	1.544154
1000000000	8	2.049445
2000000000	16	3.016202

Table 7: Weak scaling using pivot strategy 3

Data Size	Number of Processors	Normalised time
125000000	1	1.000000
250000000	2	1.158715
500000000	4	1.474925
1000000000	8	2.371577
2000000000	16	4.520712

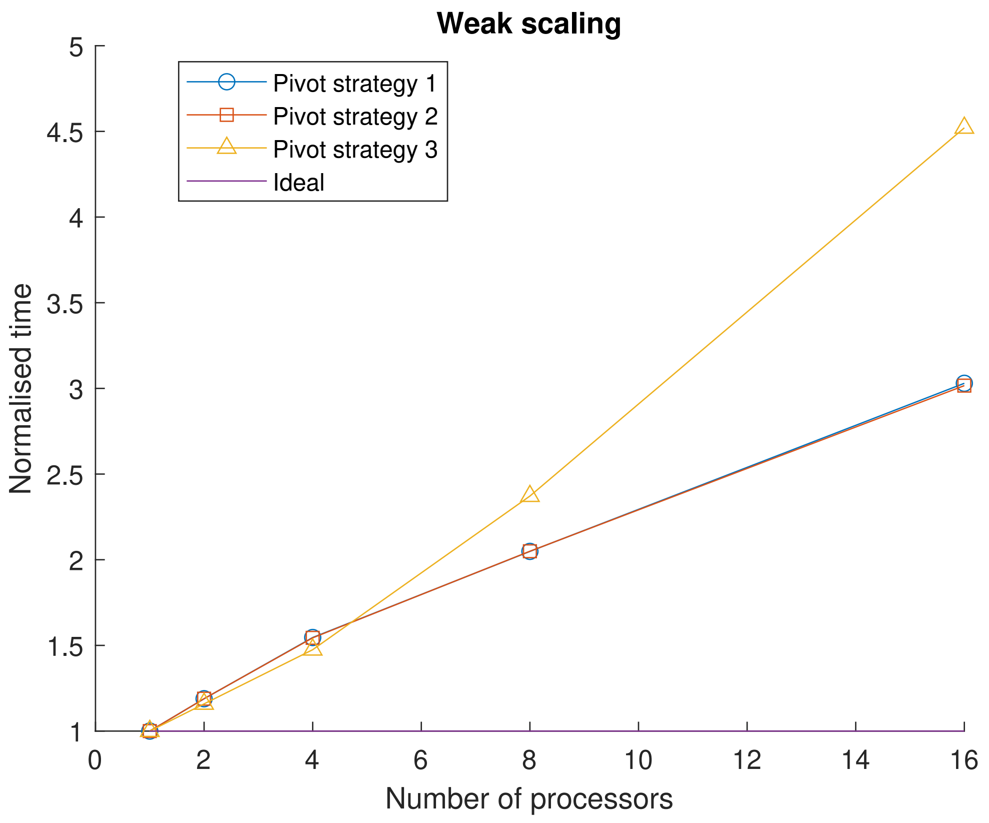


Figure 2: Weak scaling on array with data in descending order for different pivot strategies

3.2 Strong scaling

3.2.1 Random data

Table 8: Strong scaling using pivot strategy 1

Data Size	Number of Processors	Speedup
125000000	1	1.000000
125000000	2	1.936369
125000000	4	3.690411
125000000	8	6.430385
125000000	16	10.236718
125000000	32	14.475448
250000000	1	1.000000
250000000	2	1.921648
250000000	4	3.709897
250000000	8	6.595295
250000000	16	11.067471
250000000	32	16.399162

Table 9: Strong scaling using pivot strategy 2

Data Size	Number of Processors	Speedup
125000000	1	1.000000
125000000	2	1.927773
125000000	4	3.693415
125000000	8	6.628361
125000000	16	10.30577
125000000	32	14.381003
250000000	1	1.000000
250000000	2	1.9364381
250000000	4	3.696063
250000000	8	6.618649
250000000	16	11.041422
250000000	32	16.346787

Table 10: Strong scaling using pivot strategy 3

Data Size	Number of Processors	Speedup
125000000	1	1.000000
125000000	2	1.886934
125000000	4	3.239888
125000000	8	4.845366
125000000	16	5.952464
125000000	32	6.772957
250000000	1	1.000000
250000000	2	1.864242
250000000	4	3.449635
250000000	8	4.824674
250000000	16	5.989788
250000000	32	7.133409

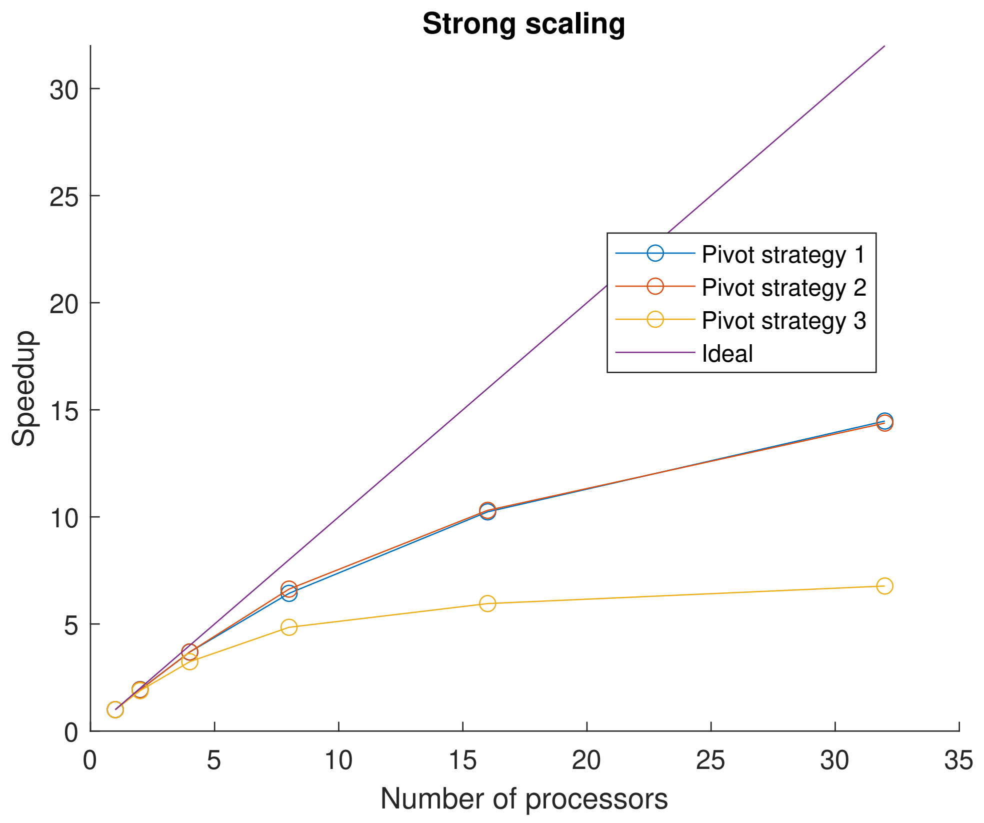


Figure 3: Strong scaling for different choices of pivot strategy on uniformly distributed random data

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	1.772890
250000000	4	3.074231
250000000	8	4.844154
250000000	16	6.915791
250000000	32	6.915038

Table 12: Strong scaling using pivot strategy 2

Data Size	Number of Processors	Speedup
250000000	1	1.000000
250000000	2	1.754051
250000000	4	3.123018
250000000	8	4.938988
250000000	16	7.041674
250000000	32	8.081750

Table 13: Strong scaling using pivot strategy 3

Data Size	Number of Processors	Speedup
250000000	1	1.000000
250000000	2	1.804422
250000000	4	3.186906
250000000	8	5.322730
250000000	16	7.080543
250000000	32	7.357109

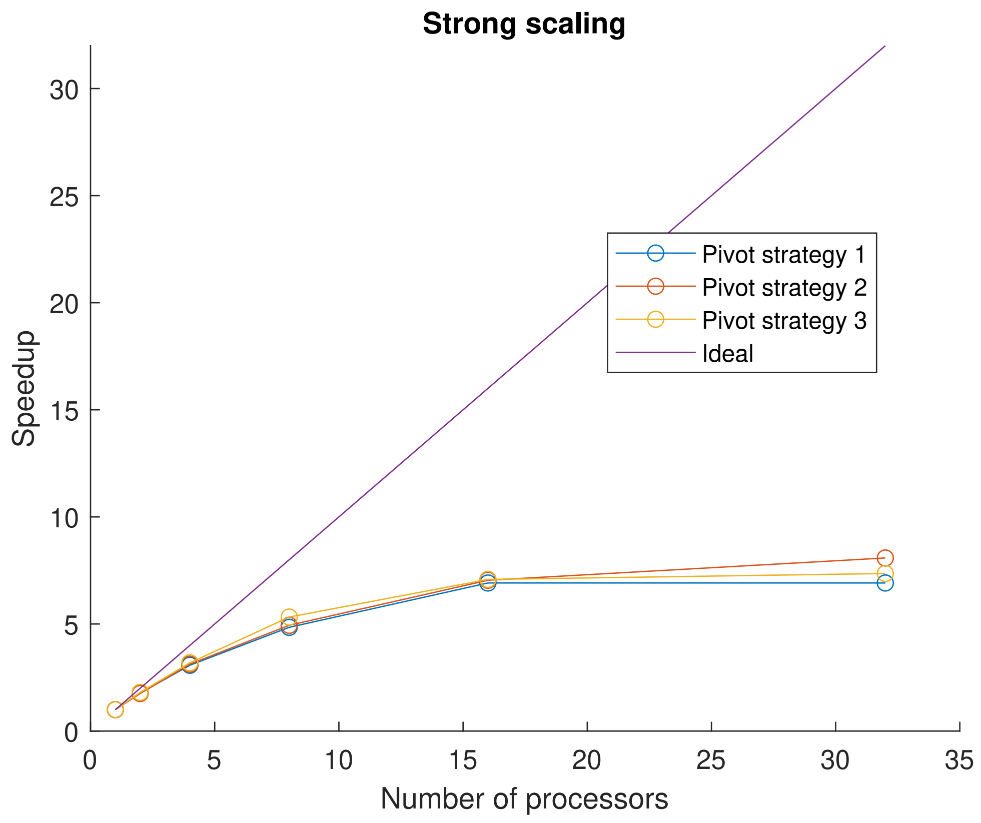
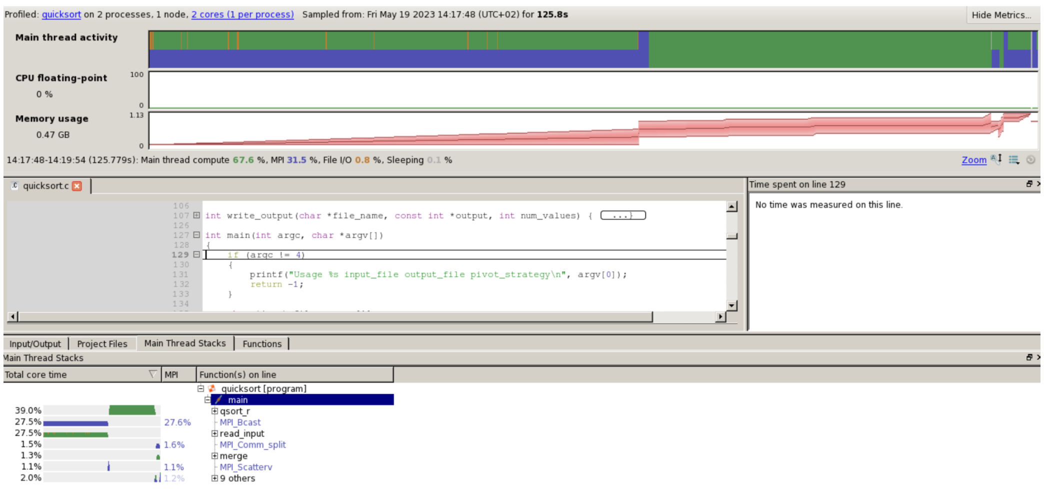


Figure 4: Strong scaling on array with data in descending order for different pivot strategies

3.3 Profiling



```
Profiled: quicksort on 2 processes, 1 node, 2 cores (1 per process)   Sampled from: Fri May 19 2023 14:17:48 (UTC+02) for 125.8s   Hide Metrics...
Main thread activity
CPU floating-point  0 %
Memory usage  0.47 GB
14:17:48-14:19:54 (125.779s): Main thread compute 67.6 %, MPI 31.5 %, File I/O 0.8 %, Sleeping 0.1 %
```

quicksort.c

```c
int write_output(char *file_name, const int *output, int num_values) { ...}

int main(int argc, char *argv[])
{
    if (argc != 4)
    {
        printf("Usage %s input_file output_file pivot_strategy\n", argv[0]);
        return -1;
    }
```

Time spent on line 129

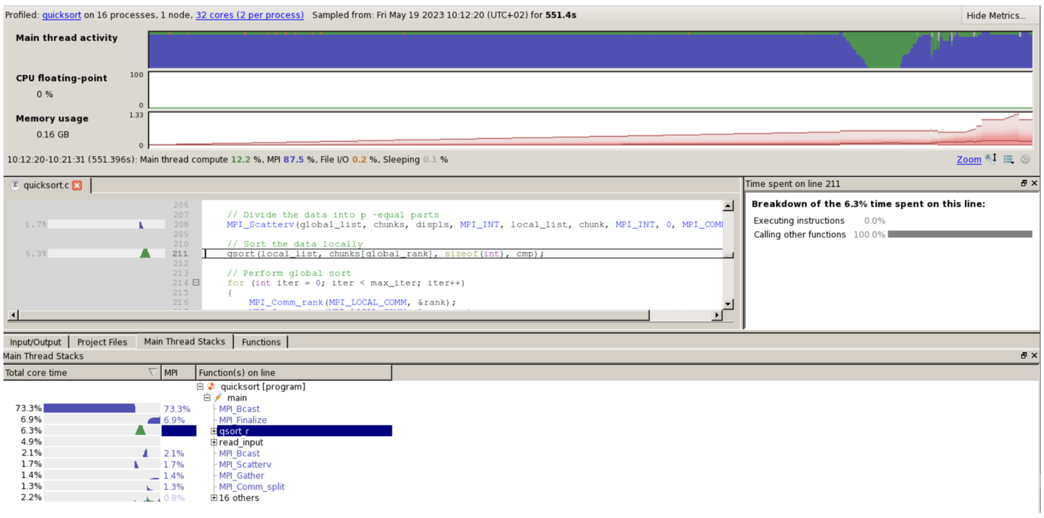
No time was measured on this line.

Input/Output | Project Files | Main Thread Stacks | Functions

Main Thread Stacks

```
Total core time   MPI     Function(s) on line
                          quicksort [program]
                            main
39.0%                         qsort_r
27.5%             27.6%       MPI_Bcast
27.5%                         read_input
 1.5%              1.6%       MPI_Comm_split
 1.3%                         merge
 1.1%              1.1%       MPI_Scatterv
 2.0%              1.2%       9 others
```

Figure 5: Profiling using pivot strategy 3, 125 000 000 elements and 2 processing units



```
Profiled: quicksort on 16 processes, 1 node, 32 cores (2 per process)   Sampled from: Fri May 19 2023 10:12:20 (UTC+02) for 551.4s   Hide Metrics...
Main thread activity
CPU floating-point  0 %
Memory usage  0.16 GB
10:12:20-10:21:31 (551.396s): Main thread compute 12.2 %, MPI 87.5 %, File I/O 0.2 %, Sleeping 0.1 %
```

quicksort.c

```c
    // Divide the data into p ~equal parts
1.7%    MPI_Scatterv(global_list, chunks, displs, MPI_INT, local_list, chunk, MPI_INT, 0, MPI_COM

    // Sort the data locally
6.3%    qsort(local_list, chunks[global_rank], sizeof(int), cmp);

    // Perform global sort
    for (int iter = 0; iter < max_iter; iter++)
    {
        MPI_Comm_rank(MPI_LOCAL_COMM, &rank);
```

Time spent on line 211

Breakdown of the 6.3% time spent on this line:

Executing instructions	0.0%
Calling other functions	100.0%

Input/Output | Project Files | Main Thread Stacks | Functions

Main Thread Stacks

```
Total core time   MPI     Function(s) on line
                          quicksort [program]
                            main
73.3%             73.3%       MPI_Bcast
 6.9%              6.9%       MPI_Finalize
 6.3%                         qsort_r
 4.9%                         read_input
 2.1%              2.1%       MPI_Bcast
 1.7%              1.7%       MPI_Scatterv
 1.4%              1.4%       MPI_Gather
 1.3%              1.3%       MPI_Comm_split
 2.2%              0.8%       16 others
```

Figure 6: Profiling using pivot strategy 3, 125 000 000 elements and 16 processing units

4 Discussion

Regarding the experiments in general, the parallel results are mediocre. The best performing metric is the strong scaling.

One reason for the speedup not being perfectly linear is that the number of iterations done is being scaled with $log_2(num_proc)$ where num_proc is the number of processors used. Therefore it is not possible to reach a perfect linear speedup.

The findings of the weak scaling experiments demonstrate that the program's runtime scales upwards with increasing numbers of processors. Using 16 processors the runtime is about twice as high as when using 1 processor. Achieving an ideal scaling is a very difficult task and these results are not that bad. Pivot strategy number 3 performs worse than the other two strategies.

Comparing the results depicted in Figures 1 and 2 indicate that the performance of all pivot strategies varies significantly when sorting random versus descending order data. It is postulated that this discrepancy arises due to a combination of load balancing issues stemming from the choice of the pivot value and parallel overhead, mainly arising from communication between the processes. Here again pivot strategy number 3 performs the worst.

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 size of the list 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 can be attributed to suboptimal load balancing. 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. This is discussed further in the paragraph below. 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. Using the MAP profiler on Uppmax, it was observed that MPI communication accounted for a larger proportion of the program's runtime when compared to a smaller number of processes. The MAP profiler was executed on 2 and 16 processes, revealing that the communication overhead was significantly higher for the latter case. This discrepancy in communication efficiency provides an explanation for the observed non-ideal speedup. The result from the profiler can be seen in 6 which shows that out of the 10.1% of the total time sorting the list, almost 38% of the time is spent in MPI calls. Compared to 2 processes where in 5 it can be seen that only 7% of the total time sorting the list is spent in MPI communication.

Looking at the strong scaling for the backwards sorted data seen in figure 4, there are some remarkable results. 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. Pivot strategies 1 an 2 performed very similar in most of the experiments. Pivot strategy number 3 performed worse than the other strategies for all experiments except for one. For strong scaling with backwards sorted elements the strategies all performed the same. The reason for pivot strategy 3 performing the worst could be of a couple of reasons. Calculating the mean value of all medians lead to more communication between the processors, thus leading to a larger overhead. I could also be of pivot bias of the mean value not being a good representation of the distribution, and could lead to load imbalances between the processors. This was closer analyzed by plotting the chunk sizes for each rank in each iteration. For 16 used processors, rank $0-7$ received 0 elements after the first iterations, practically terminating them and reducing the number of working processes to 8. This is of course very load imbalanced. The first half of the working processes consistently received very few elements during the following iterations, resulting in the last processor doing all computations in the last iteration. This might be due to unlucky pivot choices where pivot = $\sum_{medians}$/num_proc happens to be zero or very small. It shall be noted that pivot strategy number 3 performs the best on uniformly random distributed data on large in the range $[0\frac{N}{100}]$ where the risk of the pivot element being 0 is reduced drastically. The same chunk size analysis was done for the other strategies, and here the load imbalance was non existing.

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, pivot strategy number 3 performed once again the worst.

To summarize, this implementation of the parallel quicksort algorithm is performing okay on both the weak and strong scaling metrics. The choice of pivot strategy will effect the total time taken. Pivot strategy number 3 is the worst performing strategy because of the previous discussed load imbalances. Before strategy number 3 will be ruled to be the worst it has to be noted that it has only been analyzed on data in descending order and of unknown distributions, and might perform better than the others on known distributions. More work needs to be done on more distributions of data, some examples are normal and exponential distributions.