Parallel and Distributed Programming – Assignment 3
Group 15

Claude Carlsson
Edvin Germer
Ture Hassler

May 7, 2023

UPPSALA
UNIVERSITET

Contents

1 Introduction

In this assignment, we are going to implement a parallelized version of the popular sorting algorithm Quicksort. In this implementation we will experiment with three different pivot strategies, the median of the processor with rank zero, the median of all processors, and the mean of the medians of all processors. Quick sort is a sorting algorithm that divides an array into sub-arrays based on a pivot element, then it is recursively sorting each of the sub-arrays. The time complexity of Quicksort is $\mathcal{O}(n \log n)$ on average, but it performs $\mathcal{O}(n^2)$ in the worst case. There are multiple ways of choosing a pivot element in for the Quicksort algorithm, and when running in parallel this is even more important since it also handles load balancing.

We will conduct numerous tests in order to evaluate the performance of our implementation. First we will perform a strong scaling test to see how efficient our algorithm is, this will be done by keeping the problem size fixed while we increase the number of processes.

Then we will perform weak scaling tests where we increase the difficulty size together with the amount of processes in proportion to our algorithms time complexity. Furthermore, we will examine the performance of the three pivot strategies in order to determine their relative effectiveness.

2 Parallelization

2.1 Algorithm implementation

We have parallelized and implemented quick sort by following this algorithm:

1. Divide the data into p equal parts, one per processes
2. Sort the data locally for each process
3. Perform global sort
   (a) Select pivot element
   (b) Locally in each process, divide the data into two sets according to the pivot
   (c) 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.
   (d) Merge the two sets of numbers in each process into one sorted list
4. Repeat (3.a) - (3.d) recursively for each half until each group consists of one single process.

(1.) First, we read the data into **big_list**, the data is then evenly distributed among all processes using **MPI_Scatterv**. This is required instead of only **MPI_Scatter** to handle the case when the list is not evenly divisible between processes.

(2.) Then, the data is sorted locally for each process by calling the function **quicksort** with its local list and the length of the local list.

(3.) After the data has been sorted locally, we call the function **par_quicksort** in order for us to begin the global sort. (3.a) The first thing we do in this function so to select the pivot element, which we do in the function **select_pivot**. See subsection below. (3.b) Then, after we have selected the pivot element, we divide the data into two sets by using a for loop. (3.c) In order for us to split the processes into two groups and exchange data pairwise we use **MPI_Send** and **MPI_Recv**, first we exchange the length of the lists then we exchange the data. (3.d) Then we merge the two sets of numbers in each process into one sorted list by calling the function **merge**. It utilises the fact that we know that both parts of the list is already sorted, so only the overlapping parts needs to be sorted together.

(4.) Finally, steps (3.a) to (3.d) is repeated by recursively calling the function **par_quicksort** from inside the same function **par_quicksort**. We do this by splitting the communicator into two groups, using **MPI_Comm_Split**.

Then we gather all the results in main from all the sublists by using **MPI_Gather**.

2.2 Pivot strategies

We have implemented three different pivot strategies:

1. Selecting the median of a single process in each group. This was implemented by having process 0 find their median value and then broadcast it to the other processes in the communcations group.
2. Selecting the median of all medians in each group. Each processor calculated the median value of their local list, and sent it to process 0 of that respective group using **MPI_Gather**. Process 0 then selects the median of all medians and and sends it back using **MPI_Bcast**.
3. Selecting the mean of all medians in each group. It was implemented similar to the pivot strategy above but calculated the mean value of all medians instead.

3 Performance experiments and results

For the performance experiments we performed measurements of both the strong and the weak scaling, and in addition to this we also measured the memory usage.

3.1 System specifications

All timed measurements were performed on the UPPMAX high performance computing centre.

System: Uppmax Cluster "snowy"
Processor: Intel Xeon E5-2660
Operating System: CentOS Linux 7 (Core)
Compiler version: gcc 12.2.0
MPI version: openmpi 4.1.4

3.2 Performance experiments

In our experiments, we are going to evaluate both strong scaling and weak scaling. For strong scaling we are going to use a problem size of 125 million numbers, and we will test on 1, 2, 4, 8 and 16 processes.

For the weak scaling we tried to keep the work per processing element constant. Since the algorithm is $\mathcal{O}(n \log n)$, we need to adjust our problem size for weak scaling accordingly. In our results, pivot 1 refers to the strategy where we choose the median value of the sub-array for the processor of rank 0, pivot 2 refers to the median of medians, and pivot 3 refers to the strategy where we choose the means of the medians.

3.3 Results

3.3.1 Strong scaling

Table 1: The time measured for the different pivot strategies and ideal time

Number of processors	Pivot 1	Pivot 2	Pivot 3	Ideal time
1	15,81	15,83	15,83	15,83
2	9,09	9,11	9,11	7,91
4	4,92	6,13	5,93	3,96
8	3,20	4,68	4,68	1,98
16	4,01	4,09	4,07	0,99

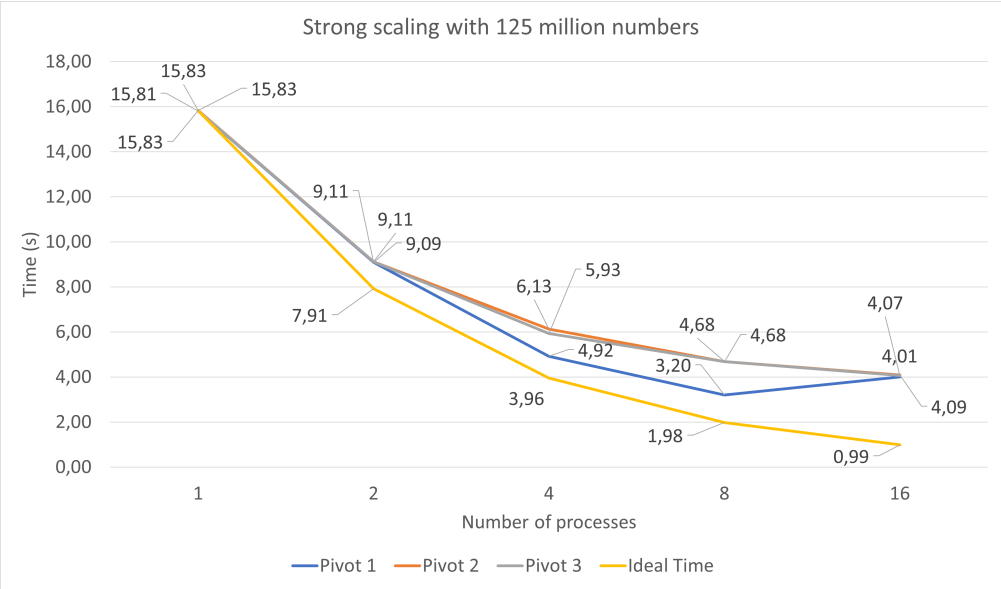

Figure 1: Strong scaling result, with measurements for each pivot strategy and ideal time

Table 2: The speedup for strong scaling with 125 million numbers

Number of processors	Pivot 1	Ideal time
1	1,0	1,0
2	1,7	2,0
4	3,2	4,0
8	4,9	8,0
16	3,9	16,0

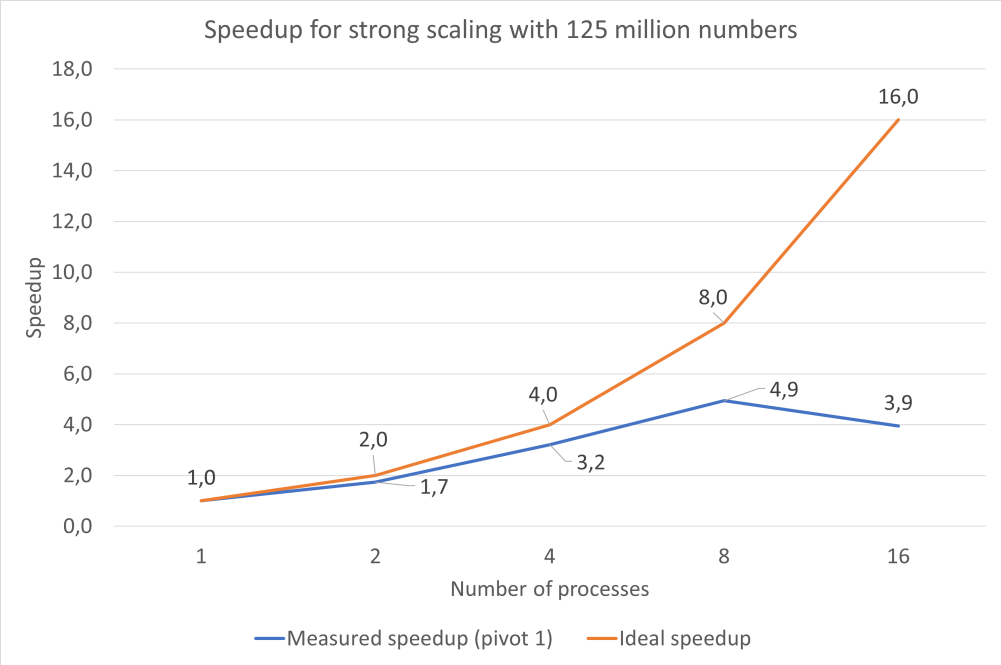

Figure 2: Speedup for strong scaling using pivot 1 and ideal time

3.3.2 Weak scaling

Table 3: The time measured for the different pivot strategies and ideal time, 125M refers to 125 million numbers

Number of processors & size	Pivot 1	Pivot 2	Pivot 3	Ideal time
1 & 125M	15,81	15,72	15,82	15,82
2 & 250M	17,65	18,97	18,78	15,82
4 & 500M	24,99	25,44	25,28	15,82
8 & 1000M	29,82	39,48	39,42	15,82

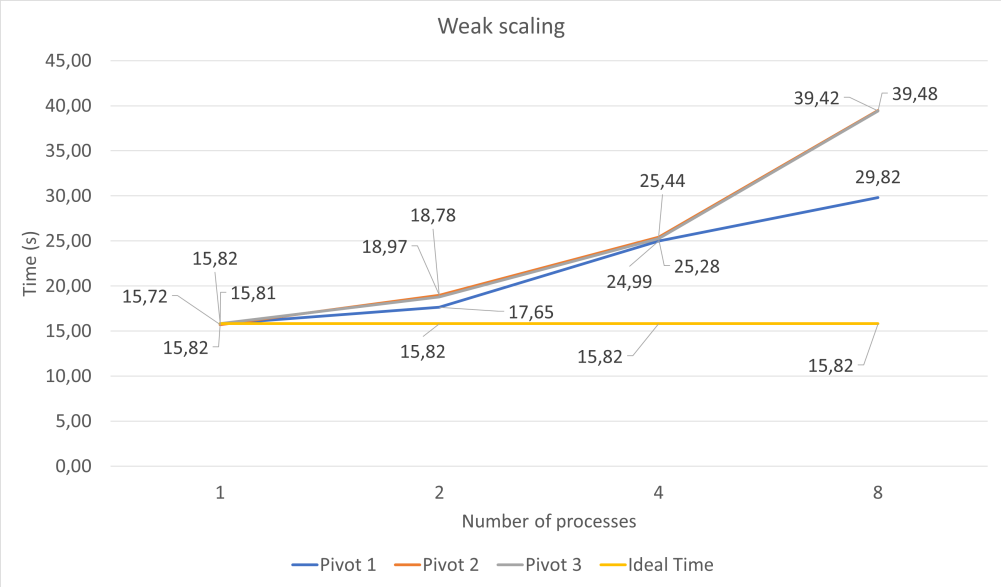

Figure 3: Speedup for strong scaling using pivot 1 and ideal time

3.4 Memory usage

After we ran the application with problem size 125 million numbers with 8 processes, we examined how much memory the job used on UPPMAX. By running:

```
1 sacct -j <job_id> --format=JobID,JobName,MaxRSS
```

we can see what our peak memory usage was during the execution of the job. Our job used approximately 117,896KB.

4 Discussion and analysis

Based on our experiments, we can see that we achieved relatively good scaling, especially up to eight processes. Looking at figure 1 we can see that the first pivot strategy achieved closest to optimal scaling but got worse after eight processes. In theory, since the first pivot strategy only takes the first processes median into account, it is more susceptible to choosing a bad pivot element. However, since it does not involve the other processes to calculate a pivot element, it reduces the communication time, which is why it performs the best out of the three different pivot strategies. However this is highly dependent on the distribution of the data. Furthermore, for weak scaling, see figure 3, we can see that the first pivot strategy also performs the best, and the reasons are probably the same as they wore for strong scaling.

Looking at figure 1 and 3 we can see that the pivot strategy 2 and 3 almost performs exactly the same. One reason for this is probably that the distribution of the input data is very even, which results in that the pivot elements for strategy 2 and 3 are very close, if not the same, most of the time.

4.1 Future improvements

The current solution reads the input data on the root rank, while the other ranks are idle. This is inefficient and takes up a large portion of the total execution time. This task of reading the data is a parallelizable task and a possible improvement would be that the root rank reads the total number of elements, distribute the indices of where each ranks section of the data starts, and then each rank would read its own data directly. Ideally, this would reduce the reading of the data as $\frac{1}{N}$ which would be significant in our case.

In order to deploy our solution effectively for real applications it would be beneficial to know where the input data comes from. This would give us an understanding of how the data will be distributed, which would allow us to select the optimal pivot strategy for that specific application.